UNLIMITED

313435

(3)

(2)

AD-A257 930

DEFENCE RESEARCH AGENCY
MALVERN

DTIC
ELECTE
NOV 24 1992
S A D

MEMORANDUM No. 4629

EXECUTABLE TRANSFORMATIONAL RULES FROM CORE ELLA TO THE KERNEL

Authors: M G Hill & J D Morison

92-30088

MEMORANDUM No. 4629

DEFENCE RESEARCH AGENCY,
MALVERN,
WORCS.

92 11 23 034

UNLIMITED

CONDITIONS OF RELEASE

0134069 313435

************************** DRIC U

************************** DRIC Y

Reports quoted are not necessarily available to members of the public or to commercial organisations.

DEFENCE RESEARCH AGENCY
MALVERN

MEMORANDUM 4629

Title: EXECUTABLE TRANSFORMATIONAL RULES FROM CORE ELLA TO THE KERNEL

Authors: M G Hill, J D Morison

Date: July 1992

Summary

This document describes the set of formal transformation rules which have been implemented for mapping Core ELLA into Kernel data structures. Examples are given of circuits which been successfully transformed by the implementation.

Intentionally Blank

Contents

1 Introduction

In this memorandum we give the formal definition of the Lisp implementation of the transformational rules from Core ELLA to the **Kernel**. In the document [MH91] a formal description was given of transformational rules from Core to **Kernel**. These rules were then implemented in the language Lisp. In the implementation of these rules a number of revisions were undertaken, the most profound being the way in which the transformational environment was handled and in the updating of the scoping rules. This has meant that the definition of a significant number of the transformational rules have changed. It is the purpose of this document to formally describe the actual functions and transformational rules which have been implemented. In order to aid readability the rules will be specified in the VDM notation [Jon90]. The reader is referred to [MH91] for a complete description of the background to this document.

This memorandum describes the static semantics for the Core of the latest version of ELLA, namely ELLA2000. In order to get the complete description of the Core language the dynamic semantics must also be considered. In reference [Hil92] a description of the dynamic semantics of the **Kernel** is given and interested readers are referred to that document.

2 The Kernel

The **Kernel** is a set of data structures into which any Core ELLA description can be mapped, for a complete description of Core ELLA and its relation with the **Kernel** the reader is referred to [MH91]. A Glossary of symbols, the definitions of Core ELLA, and the **Kernel** data structures are given in appendices A, B and C, with the transformational signatures given in appendix D.

Central to the transformation of Core ELLA descriptions is the transformational environment. For completeness the definition of the environment is given in appendix E. In the work described in [MH91] the environment appeared within the transformational rules. In the work described here the environment is updated through side-effects and thus only the following need to be considered.

Env	(the current environment)
Envstack	(a stack of environments)

The **Envstack** hold all the necessary environments when local declarations are encountered.

When beginning translation the initial environment contains only empty declarations i.e.

$$\mathit{InitialEnv} = ([\,],[\,],[\,],\{\,\},\{\,\},\{\,\},\{\,\},\{\,\},\{\,\}\{\,\},\{\,\},\{\,\}) \in \mathit{Env}$$

3 Scopes

The scopes of Core ELLA are removed by the transformation to the **Kernel**. In order to achieve this the following operators are needed

The setting of the global environment

$SetEnv(e\colon Env)b\colon \mathbf{B}$

ext wr $\mathbf{Env}\colon Env$

post $b \Leftrightarrow \mathbf{Env} = e$

The stacking of the global environment onto the local environment stack

$Stackenv()b\colon \mathbf{B}$

ext rd $\mathbf{Env}\colon Env$
wr $\mathbf{Envstack}\colon EnvStack$

post $b \Leftrightarrow \mathbf{Envstack} = [\,\mathbf{Env}] \frown \mathbf{Envstack}$

The unstacking of the local environment stack

$Unstackenv()b\colon \mathbf{B}$

ext wr $\mathbf{Envstack}\colon EnvStack$

post $b \Leftrightarrow \mathbf{Envstack} = \mathsf{tl}\ \mathbf{Envstack}$

The obtaining of the last local environment to be stacked

$Hdstack()e\colon Env$

ext rd $\mathbf{Envstack}\colon EnvStack$

post $e = \mathsf{hd}\ \mathbf{Envstack}$

We now present operators which define the environment when entering and leaving declarations.

$Scope\text{-}Fn\text{-}Begin()$

ext rd $\mathbf{Env}\colon Env$

pre $Stackenv$

post $SetEnv(\,\mathbf{env}(\ \mathbf{Env}.typedec,\ \mathbf{Env}.fndec,\ [\,],$
$\mathbf{Env}.fnmap \dagger \mathbf{Env}.lclfnmap,\ \{\,\},$
$\mathbf{Env}.tynamemap \dagger \mathbf{Env}.lcltynamemap,\ \{\,\},$
$\{\,\},\ \{\,\},$
$\{\,\},\ \{\,\},\ \{\,\}))$

Scope-Fn-End()

ext rd **Env**: *Env*

post *SetEnv*(**env**(**Env**.*typedec*, **Env**.*fndec*, (**Hdstack**).*sigdec*,
(**Hdstack**).*fnmap*, (**Hdstack**).*lclfnmap* † **Env**.*lclfnmap*,
(**Hdstack**).*tynamemap*, (**Hdstack**).*lcltynamemap* † **Env**.*lcltynamemap*,
(**Hdstack**).*signamemap*, (**Hdstack**).*lclsignamemap*,
(**Hdstack**).*usedtyname*, (**Hdstack**).*usedfnname*, (**Hdstack**).*usedsigname*))

∧ *Unstackenv*

Scope-Begin()

ext rd **Env**: *Env*

pre *Stackenv*

post *SetEnv*(**env**(**Env**.*typedec*, **Env**.*fndec*, **Env**.*sigdec*,
Env.*fnmap* † **Env**.*lclfnmap*, { },
Env.*tynamemap* † **Env**.*lcltynamemap*, { },
Env.*signamemap* † **Env**.*lclsignamemap*, { }
{ }, { }, { }))

Scope-End()

ext rd **Env**: *Env*

post *SetEnv*(**env**(**Env**.*typedec*, **Env**.*fndec*, **Env**.*sigdec*,
(**Hdstack**).*fnmap*, (**Hdstack**).*lclfnmap*,
(**Hdstack**).*tynamemap*, (**Hdstack**).*lcltynamemap*,
(**Hdstack**).*signamemap*, (**Hdstack**).*lclsignamemap*,
(**Hdstack**).*usedtyname* (**Hdstack**).*usedfnname* (**Hdstack**).*usedsigname*))

∧ *Unstackenv*

Scope-End-Add-Fn()*b*: **B**

ext rd **Env**: *Env*

post *b* ⇔
SetEnv(**env**(**Env**.*typedec*, **Env**.*fndec*, (**Hdstack**).*sigdec*,
(**Hdstack**).*fnmap*, (**Hdstack**).*lclfnmap*,
(**Hdstack**).*tynamemap*, (**Hdstack**).*lcltynamemap*,
(**Hdstack**).*signamemap*, (**Hdstack**).*lclsignamemap*,
(**Hdstack**).*usedtyname* (**Hdstack**).*usedfnname* (**Hdstack**).*usedsigname*))

$$\mathit{Local\text{-}Scope\text{-}Rule} : \mathit{Name} \to \mathbb{B}$$

$$\begin{array}{l}
\mathit{Local\text{-}Scope\text{-}Rule}(\mathit{name}) \quad \triangleq \\
\quad \textsf{if } \mathit{name} \in \textbf{Env}.\mathit{signamemap} \\
\quad \textsf{then } \mathit{SetEnv}(\mu(\textbf{ Env}, \{\mathit{usedsigname} \mapsto \textbf{ Env}.\mathit{usedsigname} \dagger \{\mathit{name}\}\})) \\
\quad \textsf{else if } \mathit{name} \in \textbf{ Env}.\mathit{tynamemap} \\
\quad\quad \textsf{then } \mathit{SetEnv}(\mu(\textbf{ Env}, \{\mathit{usedtyname} \mapsto \textbf{ Env}.\mathit{usedtyname} \dagger \{\mathit{name}\}\})) \\
\quad\quad \textsf{else if } \mathit{name} \in \textbf{ Env}.\mathit{fnmap} \\
\quad\quad\quad \textsf{then } \mathit{SetEnv}(\mu(\textbf{ Env}, \{\mathit{usedfnname} \mapsto \textbf{ Env}.\mathit{usedfnname} \dagger \{\mathit{name}\}\})) \\
\quad\quad\quad \textsf{else true}
\end{array}$$

The stacking and unstacking of the scopes for BEGIN..END clauses is carried out through the transformation rule [CC3]. Whilst the stacking and unstacking of local function and type declarations are carried out through the transformation rules [SP1] and [SP2]. The Local Scope Rule ensures that any name only has one meaning per scope.

4 Transformational Functions

In this section we present functions which will be used by the transformational rules.

4.1 Join Checks

The *Check-Joins* predicate is used to ensure that all local signals in an Environment have been joined.

$$\mathit{Check\text{-}Joins} : \emptyset \to \mathbb{B}$$

$$\mathit{Check\text{-}Joins}() \quad \triangleq \quad \forall s \in \textsf{rng } \textbf{Env}.\mathit{lclsignamemap} \cdot s.\mathit{sort} = \textbf{ joined}$$

4.2 Two Value Types

Here we present the predicate for checking that a type is a two valued enumerated type:

$$\mathit{Check\text{-}Two\text{-}Val} : \mathit{kType} \to \mathbb{B}$$

$$\begin{array}{l}
\mathit{Check\text{-}Two\text{-}Val}(\mathit{ty}) \quad \triangleq \\
\quad \textsf{let } \textbf{typeno}(\mathit{typeno}) = \mathit{ty} \textsf{ in} \\
\quad \textsf{let } (\textbf{ Env}.\mathit{typedec})[\mathit{typeno}].\mathit{new} = \textbf{ tags}(\mathit{TagSeq}) \textsf{ in} \\
\quad\quad \textsf{len } (\textbf{ tags}(\mathit{TagSeq})) = 2
\end{array}$$

4.3 Check names

These predicates ensure that a particular name is not already in scope. They will be used by the functions that add names to an Environment.

$Check\text{-}Fn\colon Fnname \rightarrow \mathbb{B}$

$Check\text{-}Fn(fnname) \triangleq$
$\quad fnname \notin (\mathrm{dom}(\mathbf{Env}.lclfnmap) \cup \mathbf{Env}.usedfnname)$

$Check\text{-}Typename\colon Name \rightarrow \mathbb{B}$

$Check\text{-}Typename(name) \triangleq$
$\quad name \notin (\mathrm{dom}(\mathbf{Env}.lcltynamemap) \cup \mathbf{Env}.usedtyname)$

$Check\text{-}Signal\colon Signalname \rightarrow \mathbb{B}$

$Check\text{-}Signal(signalname) \triangleq$
$\quad signalname \notin (\mathrm{dom}(\mathbf{Env}.lclsignamemap) \cup \mathbf{Env}.usedsigname)$

4.4 Adding Names to an Environment

These functions define the addition of names to an environment

$Add\text{-}Fn\colon Fndec \rightarrow \mathbb{B}$

$$
\begin{array}{l}
Add\text{-}Fn(fd) \triangleq \\
\quad \mathsf{let}\ Len = \mathsf{len}\ \mathbf{Env}.fndec\ \mathsf{in} \\
\quad \mathsf{let}\ FnName = fd.fnname\ \mathsf{in} \\
\qquad SetEnv(\mu(\mathbf{Env}, \{ fndec \mapsto \mathbf{Env}.fndec \frown [fd], \\
\qquad\qquad\qquad\qquad lclfnmap \mapsto (\mathbf{Env}.lclfnmap \dagger \{FnName \mapsto Len + 1\}) \\
\qquad\qquad\qquad \})) \\
\mathsf{pre}\ Check\text{-}Fn(fd.fnname) \wedge Scope\text{-}End\text{-}Add\text{-}Fn
\end{array}
$$

$Add\text{-}Type\colon Typedec \rightarrow \mathbb{B}$

$$
\begin{array}{l}
Add\text{-}Type(td) \triangleq \\
\quad \mathsf{let}\ Len = \mathsf{len}\ \mathbf{Env}.typedec\ \mathsf{in} \\
\quad \mathsf{let}\ TyName = td.typename\ \mathsf{in} \\
\qquad SetEnv(\mu(\mathbf{Env}, \{ typedec \mapsto \mathbf{Env}.typedec \frown [td], \\
\qquad\qquad\qquad\qquad lcltynamemap \mapsto \\
\qquad\qquad\qquad\qquad\qquad (\mathbf{Env}.lcltynamemap \dagger \{TyName \mapsto \mathbf{typeno}(Len + 1)\}) \\
\qquad\qquad\qquad \})) \\
\mathsf{pre}\ Check\text{-}Typename(td.typename) \wedge Check\text{-}Signal(td.typename)
\end{array}
$$

Add-Signal: *Signaldec* × *Sort* → **B**

Add-Signal(*sd*, *sort*) $\underline{\triangle}$
  let *Len* = len **Env**.*sigdec* in
  let *SigName* = *sd*.*signalname* in
    SetEnv(μ(**Env**, { *sigdec* ↦ **Env**.*sigdec* ⌢ [*sd*],
      lclsignamemap ↦
        (**Env**.*lclsignamemap* † {*SigName* ↦ **sig**(*Len* + 1, *sort*)})
    }))
pre *Check-Signal*(*sd*.*signalname*) ∧ *Check-Typename*(*sd*.*signalname*)

Add-Join: *Signaldec* × *Signalno* → **B**

Add-Join(*sd*, *signalno*) $\underline{\triangle}$
  let *SigName* = *sd*.*signalname* in
    SetEnv(μ(**Env**, { *sigdec*[*signalno*] ↦ *sd*,
      lclsignamemap ↦ (**Env**.*lclsignamemap* † {*SigName* ↦
        sig(*signalno*, **joined**)})
    }))

Add-Tag: *Tagname* → **B**

Add-Tag(*tagname*) $\underline{\triangle}$
  let *Len* = len **Env**.*typedec* in
    SetEnv(μ(**Env**, { *lcltynamemap* ↦
        (**Env**.*lcltynamemap* † {*tagname* ↦ **consttag**(*Len* + 1)})
    }))
pre *Check-Typename*(*tagname*) ∧ *Check-Signal*(*tagname*)

Add-Type-Name: *Typename* × *kType* → **B**

Add-Type-Name(*typename*, *ktype*) $\underline{\triangle}$
  SetEnv(μ(**Env**, { *lcltynamemap* ↦
      (**Env**.*lcltynamemap* † {*typename* ↦ **typename**(*typename*, *ktype*)})
  }))
pre *Check-Typename*(*typename*) ∧ *Check-Signal*(*typename*)

4.5 Finding Names in an Environment

These functions describe how any name can be located within an Environment

Find-Lower-Nm: *Name* → *Typetag* ∪ *Sig*

Find-Lower-Nm(*name*) $\underline{\triangle}$
  (**Env**.*signamemap*† **Env**.*tynamemap*† **Env**.*lclsignamemap*† **Env**.*lcltynamemap*)(*name*)
pre *Local-Scope-Rule*(*name*)

$Find\text{-}Type\text{-}or\text{-}Alt$: $Name \rightarrow kType \cup kEnum$

$Find\text{-}Type\text{-}or\text{-}Alt(name) \triangleq$
  let $ans = Find\text{-}Lower\text{-}Nm(name)$ in
  if $ans = \mathbf{consttag}(\_)$
  then $Find\text{-}Alt(name)$
  else $Find\text{-}Type(name)$

$Find\text{-}Sig\text{-}or\text{-}Alt$: $Name \rightarrow (kUnit \times kType) \cup kEnum$

$Find\text{-}Sig\text{-}or\text{-}Alt(name) \triangleq$
  let $ans = Find\text{-}Lower\text{-}Nm(name)$ in
  if $ans = \mathbf{sig}(\_, \_)$
  then $Find\text{-}Signal(name)$
  else $Find\text{-}Alt(name)$

$Find\text{-}Fn$: $Fnname \rightarrow Fnno$

$Find\text{-}Fn(name) \triangleq$
  $(\ \mathbf{Env}.fnmap \dagger \mathbf{Env}.lclfnmap)(name)$

pre $Local\text{-}Scope\text{-}Rule(name)$

$Find\text{-}Unjoined$: $Signalname \rightarrow Fnno$

$Find\text{-}Unjoined(signalname) \triangleq$
  let $Signo = (\ \mathbf{Env}.lclsignamemap)(signalname).signalno$ in
  let $\mathbf{signaldec}(signalname, type, \mathbf{instance}(fnno, \_)) = (\ \mathbf{Env}.sigdec)[Signo]$ in
   $fnno$

pre $(\ \mathbf{Env}.lclsignamemap)(signalname).sort = \mathbf{unjoined}$

$Find\text{-}Type$: $Typename \rightarrow kType$

$Find\text{-}Type(typename) \triangleq$
  $Find\text{-}Lower\text{-}Nm(typename)$

pre $Find\text{-}Lower\text{-}Nm(typename) \in (\ \mathbf{typename} \cup \mathbf{typeno})$

$Find\text{-}Alt$: $Altname \rightarrow kEnum$

$Find\text{-}Alt(altname) \triangleq$
  let $\mathbf{consttag}(ktypeno) = Find\text{-}Lower\text{-}Nm(altname)$ in
  let $\mathbf{typedec}(\_, \mathbf{tags}(tags)) = (\ \mathbf{Env}.typedec)[ktypeno]$ in
  let $index = \iota(i \in \text{inds}\ tags) \cdot tags[i] = \mathbf{tag}(altname, \_)$ in
   $\mathbf{enum}(ktypeno,\ index)$

$\mathit{Find\text{-}ELLAint}\colon \mathit{Tagname} \to \mathit{Typeno} \times \mathit{Lowerbound} \times \mathit{Upperbound}$

$\mathit{Find\text{-}ELLAint}(\mathit{tagname}) \ \underline{\triangle}$
  let $\mathbf{consttag}(\mathit{ktypeno}) = \mathit{Find\text{-}Lower\text{-}Nm}(\mathit{tagname})$ in
  let $\mathbf{typedec}(\_,\ \mathbf{ellaint}(\_, \mathit{lb}, \mathit{ub})) = (\ \mathbf{Env}.\mathit{typedec})[\mathit{ktypeno}]$ in
    $\mathit{ktypeno},\ \mathit{lb},\ \mathit{ub}$

$\mathit{Find\text{-}Integer\text{-}Type}\colon \mathit{kType} \to \mathit{Tagname} \times \mathit{Lowerbound} \times \mathit{Upperbound}$

$\mathit{Find\text{-}Integer\text{-}Type}(\mathit{ktype}) \ \underline{\triangle}$
  cases ktype of
  $\mathbf{typeno}(\mathit{typeno}) \to$ cases $(\ \mathbf{Env}.\mathit{typedec})[\mathit{typeno}]$ of
    $\mathbf{typedec}(\_,\ \mathbf{ellaint}(t, l, u)) \to t,\ l,\ u$
    end
  $\mathbf{typename}(\_, \mathit{type}) \to \mathit{Find\text{-}Integer\text{-}Type}(\mathit{type})$
  end

$\mathit{Find\text{-}Char}\colon \mathit{Tagname} \times \mathit{Char} \to \mathit{kEnum}$

$\mathit{Find\text{-}Char}(\mathit{tagname}, \mathit{char}) \ \underline{\triangle}$
  let $\mathbf{consttag}(\mathit{ktypeno}) = \mathit{Find\text{-}Lower\text{-}Nm}(\mathit{tagname})$ in
  let $\mathbf{typedec}(\_,\ \mathbf{chars}(\_, \mathit{chs})) = (\ \mathbf{Env}.\mathit{typedec})[\mathit{ktypeno}]$ in
  let $\mathit{index} = \iota(i \in \mathrm{inds}\ \mathit{chs}) \cdot \mathit{chs}[i] = \mathit{char}$ in
    $\mathbf{enum}(\mathit{ktypeno},\ \mathit{index})$

$\mathit{Find\text{-}Signal}\colon \mathit{Signal} \to \mathit{kUnit} \times \mathit{kType}$

$\mathit{Find\text{-}Signal}(\mathit{signalname}) \ \underline{\triangle}$
  let $\mathbf{sig}(\mathit{signalno}, \_) = \mathit{Find\text{-}Lower\text{-}Nm}(\mathit{signalname})$ in
    $\mathbf{signal}(\mathit{signalno}),\ (\ \mathbf{Env}.\mathit{sigdec})[\mathit{signalno}].\mathit{type}$

$\mathit{Find\text{-}Assoc}\colon \mathit{Altname} \to \mathit{kType}$

$\mathit{Find\text{-}Assoc}(\mathit{altname}) \ \underline{\triangle}$
  let $\mathbf{consttag}(\mathit{ktypeno}) = \mathit{Find\text{-}Lower\text{-}Nm}(\mathit{altname})$ in
  let $\mathbf{typedec}(\_,\ \mathbf{tags}(\mathit{tags})) = (\ \mathbf{Env}.\mathit{typedec})[\mathit{ktypeno}]$ in
    $\iota(\mathit{ktypeOpt} \in \mathit{kTypeOpt}) \cdot\ \mathbf{tag}(\mathit{altname}, \mathit{ktypeOpt}) \in \mathrm{elems}\ \mathit{tags}$

$$\textit{Find-Row}: kType \rightarrow \mathbb{N}_1 \times kType$$

$$\begin{array}{l} \textit{Find-Row}(ktype) \triangleq \\ \quad \mathsf{let}\ ty = \textit{Get-Types}(ktype)\ \mathsf{in} \\ \quad \mathsf{let}\ \mathbf{types}(types) = ty\ \mathsf{in} \\ \quad \mathsf{let}\ size = \mathsf{len}\,(types)\ \mathsf{in} \\ \quad\quad (size,\ types[1]) \\ \mathsf{pre}\ \forall i \in \{1..size\} \cdot types[1] \dashv\!\boxed{T}\!\models\ types[i] \end{array}$$

$$\textit{Find-Biop}: Biopname \times kType \times kType \rightarrow kFnbody$$

$$\begin{array}{l} \textit{Find-Biop}(name, ktype_1, ktype_2) \triangleq \\ \quad \mathsf{let}\ biopinfo = \iota(i \in \mathbf{BiopEnv}.biop) \cdot i.biopname = name\ \mathsf{in} \\ \quad\quad \mathbf{biop}(name) \\ \mathsf{pre}\ \textit{Biop-Type-Equals}(biopinfo.inputtype, ktype_1) \wedge \\ \quad \textit{Biop-Type-Equals}(biopinfo.outputtype, ktype_2) \end{array}$$

4.6 Removing Type Aliasing

Type aliasing is removed by means of the following function:

$$\textit{Get-Type} : kType \rightarrow kType$$

$$\begin{array}{l} \textit{Get-Type}(ty) \triangleq \\ \quad \mathsf{cases}\ ty\ \mathsf{of} \\ \quad\ \mathbf{types}([ktype_1, \cdots, ktype_k]) \rightarrow \mathbf{types}([\textit{Get-Type}(ktype_1), \cdots, \textit{Get-Type}(ktype_k)]), \\ \quad\ \mathbf{typename}(\_, ktype) \qquad\ \rightarrow \textit{Get-Type}(ktype) \\ \quad\ \mathbf{stringtype}(size, ktype) \quad\ \rightarrow \mathbf{stringtype}(size, \textit{Get-Type}(ktype)) \\ \quad \mathsf{others}\ ty \\ \quad \mathsf{end} \end{array}$$

4.7 Type Checking

Type checking is an important aspect of the ELLA compiler and the relation '$a \dashv\!\boxed{T}\!\models b$' shows how the transformation from Core ELLA to the Kernel will define type equality. This relation is defined by:-

$$ktype_1 \dashv\!\boxed{T}\!\models ktype_2 \quad \Leftrightarrow \quad \textit{Type-Equals}(ktype_1, ktype_2)$$

where

$\textit{Type-Equals} : \textit{kType} \times \textit{kType} \rightarrow \mathbb{B}$

$\textit{Type-Equals}(ty_1, ty_2) \triangleq$
  cases ($\textit{Get-Type}(ty_1)$, $\textit{Get-Type}(ty_2)$) **of**
  (**typeno**($typeno_1$), **typeno**($typeno_2$)) $\rightarrow (typeno_1 = typeno_2)$
  (**stringtype**(s_1, tn_1), **stringtype**(s_2, tn_2)) $\rightarrow (s_1 = s_2) \wedge \textit{Type-Equals}(tn_1, tn_2)$
  (**types**($[t_1, ..., t_k]$), **types**($[s_1, ..., s_j]$)) $\rightarrow j = k \bigwedge_{i=1..k} \textit{Type-Equals}(t_i, s_i)$
  (**typevoid**, **typevoid**) $\rightarrow$ **true**
  others false

  end

For the Built in Operators (Biops) the type of the enclosing function specification behaves like an ELLA Macro, and therefore full type checking is not possible at the static semantic stage. For that reason the following function is necessary for the type checking of a Biop specification.

$\textit{Biop-Type-Equals} : \textit{kType} \times \textit{kType} \rightarrow \mathbb{B}$

$\textit{Biop-Type-Equals}(bty_1, bty_2) \triangleq$
  cases ($\textit{Get-Type}(bty_1)$, $\textit{Get-Type}(bty_2)$) **of**
  (**typeno**($typeno$), **typeno**(_)) $\rightarrow$ **true**
  (**stringtype**(s_1, tn_1), **stringtype**(s_2, tn_2)) $\rightarrow \textit{Biop-Type-Equals}(tn_1, tn_2)$
  (**types**($[t_1, ..., t_k]$), **types**($[s_1, ..., s_j]$)) $\rightarrow j = k \bigwedge_{i=1..k} \textit{Biop-Type-Equals}(t_i, s_i)$
  (**typevoid**, **typevoid**) $\rightarrow$ **true**
  others false

  end

4.8 Type Indexing

These function describes how to obtain the type of an indexed or trimmed quantity

$\textit{Get-Index} : \textit{kType} \times \mathbb{N}_1 \rightarrow \textit{kType}$

$\textit{Get-Index}(ty, i) \triangleq$
  cases $\textit{Get-Type}(ty)$ **of**
  types($[ktype_1, \cdots, ktype_k]$) $\rightarrow ktype_i$,
  stringtype(_, $ktype$) $\rightarrow ktype$

  end

$Trim : kType \times \mathbb{N}_1 \times \mathbb{N}_1 \rightarrow kType$

$Trim(ty, lb, ub) \;\triangleq$
cases $Get\text{-}Type(ty)$ of
types$([ktype_1, \cdots, ktype_k]) \rightarrow$ **types**$([ktype_{lb}, \cdots, ktype_{ub}])$,
stringtype$(\_, ktype) \rightarrow$ **stringtype**$((ub\text{-}lb + 1), ktype)$
end

4.9 Concatenation

Concatenation of two signal types are handled by means of the following function.

$Conc : kType \times kType \rightarrow kType$

$Conc(ktype_1, ktype_2) \;\triangleq$
let $ty_1 = Get\text{-}Type(ktype_1)$ in
let $ty_2 = Get\text{-}Type(ktype_2)$ in
cases (ty_1, ty_2) of
(**types**$([ta_1, \cdots, ta_k])$,
types$([tb_1, \cdots, tb_l])$) $\rightarrow$ if $(\forall i \in \{1..k\}, j \in \{1..l\} \cdot (ta_i =\!\boxed{\mathrm{T}}\!= tb_j)) =$ true
then **types**$([ta_1, \cdots, ta_k, tb_1, \cdots, tb_l])$
else if $(\forall i \in \{1..k\} \cdot (ta_i =\!\boxed{\mathrm{T}}\!= ty_2)) =$ true
then **types**$([ta_1, \cdots, ta_k, ty_2])$
else if $(\forall j \in \{1..l\} \cdot ty_1 =\!\boxed{\mathrm{T}}\!= tb_j)) =$ true
then **types**$([ty_1, tb_1, \cdots, tb_l])$
else $\emptyset$
(**types**$([t_1, \cdots, t_k])$, _) $\rightarrow$ let $(\forall i \in \{1..k\} \cdot t_i =\!\boxed{\mathrm{T}}\!= ty_2) =$ true in
types$([t_1, \cdots, t_k, ty_2])$
(_, **types**$([t_1, \cdots, t_k])$) $\rightarrow$ let $(\forall i \in \{1..k\} \cdot t_i =\!\boxed{\mathrm{T}}\!= ty_1) =$ true in
types$([ty_1, t_1, \cdots, t_k])$
(**stringtype**$(size_a, ktype_a)$,
stringtype$(size_b, ktype_b)$) $\rightarrow$ let $(ktype_a =\!\boxed{\mathrm{T}}\!= ktype_b) =$ true in
stringtype$(size_a + size_b, ktype_a)$
(**stringtype**$(size, ktype)$, _) $\rightarrow$ let $(ktype =\!\boxed{\mathrm{T}}\!= ty_2) =$ true in
stringtype$(size + 1, ktype)$
(_, **stringtype**$(size, ktype)$) $\rightarrow$ let $(ty_1 =\!\boxed{\mathrm{T}}\!= ktype) =$ true in
stringtype$(size + 1, ktype)$
end

4.10 Reform

This function flattens types so that they are available for **reform**

$$Flatten : kType \rightarrow kTypeSeq$$

$$
\begin{array}{l}
Flatten(ty) \;\triangleq \\
\quad \textsf{cases } Get\text{-}Type(ty) \textsf{ of} \\
\quad\; \textbf{typeno}\ (typeno) \quad \rightarrow [\ \textbf{typeno}(typeno)], \\
\quad\; \textbf{stringtype}\ (size, ktype) \rightarrow [\ \textbf{stringtype}(size, ktype)], \\
\quad\; \textbf{types}\ ([t_1, \cdots, t_k]) \quad \rightarrow Flatten(t_1) \frown \cdots \frown Flatten(t_k) \\
\quad\; \textbf{typevoid} \quad \rightarrow [\ \textbf{typevoid}] \\
\quad \textsf{end}
\end{array}
$$

4.11 Character Check

This rule checks that a particular type is of the form of an ELLA character.

$$Is\text{-}Char : \mathbb{N}_1 \rightarrow \mathbb{B}$$

$$Is\text{-}Char(typeno) \;\triangleq\; (\ \mathbf{Env}.typedec)[typeno].new \in \ \mathbf{chars}$$

4.12 Constructing Tuples

These functions convert sequences into tuples.

$$TypeTuple : kTypeSeq \rightarrow kType$$

$$
\begin{array}{ll}
TypeTuple(tseq) \;\triangleq & \textsf{if len } tseq = 1 \\
 & \textsf{then } tseq[1] \\
 & \textsf{else } \textbf{types}\ (tseq)
\end{array}
$$

$$ConstTuple : kConstSeq \rightarrow kConst$$

$$
\begin{array}{ll}
ConstTuple(cseq) \;\triangleq & \textsf{if len } cseq = 1 \\
 & \textsf{then } \mathrm{cseq}[1] \\
 & \textsf{else } \textbf{consts}\ (cseq)
\end{array}
$$

$$ConstsetTuple : kConstsetSeq \rightarrow kConstset$$

$$
\begin{array}{ll}
ConstsetTuple(csseq) \;\triangleq & \textsf{if len } csseq = 1 \\
 & \textsf{then } csseq[1] \\
 & \textsf{else } \textbf{constsets}\ (csseq)
\end{array}
$$

$$UnitTuple : kUnitSeq \rightarrow kUnit$$

$$
\begin{array}{ll}
UnitTuple(useq) \;\triangleq & \textsf{if len } useq = 1 \\
 & \textsf{then } useq[1] \\
 & \textsf{else } \textbf{units}\ (useq)
\end{array}
$$

4.13 Case Disjointness

The following function checks that a CASE statement has disjoint choosers

$Disjoint : kConstset \times kConstset \rightarrow \mathbb{B}$

$Disjoint(cset_1, cset_2) \triangleq$
cases $(cset_1, cset_2)$ **of**

$(\ \mathbf{enum}(\_, tagno_1),\ \mathbf{enum}(\_, tagno_2)) \rightarrow tagno_1 \neq tagno_2$

$(\ \mathbf{string}(\_, [tagno_{11}, \cdots, tagno_{1k}]),\ \mathbf{string}(\_, [tagno_{21}, \cdots, tagno_{2k}])) \rightarrow \bigvee_{i=\{1..k\}} (tagno_{1i} \neq tagno_{2i})$

$(\ \mathbf{constsetassoc}(\ \mathbf{enum}(\_, tagno_1), constset_1),\ \mathbf{constsetassoc}(\ \mathbf{enum}(\_, tagno_2), constset_2)) \rightarrow tagno_1 \neq tagno_2 \vee Disjoint(constset_1,\ constset_2)$

$(\ \mathbf{constsets}([csa_1, \cdots, csa_k]),\ \mathbf{constsets}([csb_1, \cdots, csb_k])) \rightarrow \bigvee_{i=\{1..k\}} Disjoint(csa_i,\ csb_i)$

$(\_,\ \mathbf{constsetalts}([csa_1, \cdots, csa_k])) \rightarrow \bigwedge_{i=\{1..k\}} Disjoint(cset_1,\ csa_i)$

$(\ \mathbf{constsetalts}([csa_1, \cdots, csa_k]),\ \_) \rightarrow \bigwedge_{i=\{1..k\}} Disjoint(csa_i,\ cset_2)$

$(\ \mathbf{constsetstring}(size_a, cset_a),\ \mathbf{constsetstring}(size_b, cset_b)) \rightarrow (size_a \neq size_b) \vee Disjoint(cset_a,\ cset_b)$

$(\ \mathbf{constsetstring}(size_a, cset_a),\ \mathbf{string}(ty, [tg_1, \cdots, tg_k]) \rightarrow (size_a \neq k)\ \bigvee_{i=\{1..k\}} Disjoint(cset_a,\ \mathbf{enum}(ty, tg_i))$

$(\ \mathbf{string}(\_, \_),\ \mathbf{constsetstring}(\_, \_)) \rightarrow Disjoint(cset_2, cset_1)$

$(\ \mathbf{constsetany}(type),\ \_) \rightarrow \mathsf{false}$

$(\_,\ \mathbf{constsetany}(type)) \rightarrow \mathsf{false}$

end

4.14 Local Type Checking

$Not\text{-}Local\text{-}Type : kType \rightarrow \mathbb{B}$

$Not\text{-}Local\text{-}Type(ktype) \triangleq$
cases $Get\text{-}Type(ktype)$ **of**

$\mathbf{typeno}(typeno) \rightarrow \forall(i \in \mathrm{rng}\ \mathbf{Env}.lcltynamemap) \cdot i \neq \mathbf{typeno}(typeno)$

$\mathbf{types}([t_1, \cdots, t_n]) \rightarrow \bigwedge_{i \in \{1..n\}} Not\text{-}Local\text{-}Type(t_i)$

$\mathbf{stringtype}(size, t) \rightarrow Not\text{-}Local\text{-}Type(t)$

others true

end

This function checks that its input type is not a locally declared type, and will be used by local BEGIN..END clauses to ensure that the output from the clause only contains global types.

5 Transformational Rules

This section describes the transformational rules from Core ELLA to the **Kernel**. These include the semantic checks which are done by the full ELLA compiler on Core ELLA ie. type checking, name checking etc. Thus this section includes a description of the static semantics of Core ELLA. At the start of each subsection the Core ELLA syntax, for which the transformations of that section apply, will be given. In each rule the order of execution of the pre-conditions is left-to-right, top-to-bottom.

5.1 Enumerated Values

Enumerated values are defined by

```
enumerated   ::=   altname
                 | tagname / z
                 | tagname 'char
                 | tagname "string"
```

and the transformations on them are given by

$$\boxed{\text{EM1}}\ \frac{\textit{Find-ELLAint}\ (\textit{tagname}) = \textit{ktypeno},\ \textit{lb},\ \textit{ub} \qquad \textit{lb} \leq z \leq \textit{ul}}{[\![\textit{tagname}/z]\!] \ \boxed{\text{EM}} \Rightarrow\ \mathbf{enum}(\textit{ktypeno}, z\text{-}\textit{lb}+1) : \mathbf{typeno}(\textit{ktypeno})}$$

$$\boxed{\text{EM2}}\ \frac{\textit{Find-Char}\ (\textit{tagname}, \textit{char}) = \mathbf{enum}(\textit{ktypeno}, \textit{index})}{[\![\textit{tagname}\ '\textit{char}]\!] \ \boxed{\text{EM}} \Rightarrow\ \mathbf{enum}(\textit{ktypeno}, \textit{index}) : \mathbf{typeno}(\textit{ktypeno})}$$

$$\boxed{\text{EM3}}\ \frac{\forall i \in \{1..k\} \cdot (\textit{Find-Char}\ (\textit{tagname}, \textit{char}_i) = \mathbf{enum}(\textit{ktypeno}, \textit{ktagno}_i))}{\begin{array}{c}[\![\textit{tagname}\ "\textit{char}_1 \cdots \textit{char}_k"]\!] \ \boxed{\text{EM}} \Rightarrow \\ \mathbf{string}(\textit{ktypeno}, [\textit{ktagno}_1, \cdots, \textit{ktagno}_k]) : \mathbf{stringtype}(k,\ \mathbf{typeno}(\textit{typeno}))\end{array}}$$

5.2 Types

Types in Core ELLA can have the following form

```
type   ::=   typename
           | STRING [ size ] typename
           | [ size ] type
           | ( type1, ···, typek )
           | ()
```

and the transformations that apply to them are

$$\boxed{\text{T1}}\ \frac{\textit{Find-Type}\ (\textit{typename}) = \textit{ktype}}{[\![\textit{typename}]\!] \mathrel{=\!\boxed{\mathrm{T}}\!\Rightarrow} \textit{ktype}}$$

$$\boxed{\text{T2}}\ \frac{[\![\textit{typename}]\!] \mathrel{=\!\boxed{\mathrm{T}}\!\Rightarrow} \textit{ktype} \quad \textit{Get-Type}(\textit{ktype}) = \ \mathbf{typeno}(\textit{ktypeno}) \quad \textit{Is-Char}\ (\textit{ktypeno})}{[\![\textit{STRING}[\textit{size}]\textit{typename}]\!] \mathrel{=\!\boxed{\mathrm{T}}\!\Rightarrow} \mathbf{stringtype}(\textit{size}, \textit{ktype})}$$

$$\boxed{\text{T3}}\ \frac{[\![\textit{type}]\!] \mathrel{=\!\boxed{\mathrm{T}}\!\Rightarrow} \textit{ktype}}{[\![[\textit{size}]\textit{type}]\!] \mathrel{=\!\boxed{\mathrm{T}}\!\Rightarrow} \mathbf{types}([\textit{ktype}^{\textit{size}}])}$$

$$\boxed{\text{T4}}\ \frac{\forall i \in \{1..k\} \cdot ([\![\textit{type}_i]\!] \mathrel{=\!\boxed{\mathrm{T}}\!\Rightarrow} t_i)}{[\![(\textit{type}_1, \cdots, \textit{type}_k)]\!] \mathrel{=\!\boxed{\mathrm{T}}\!\Rightarrow} \textit{TypeTuple}([t_1, \cdots, t_k])}$$

$$\boxed{\text{T5}}\ \frac{}{[\![\ ()\ ]\!] \mathrel{=\!\boxed{\mathrm{T}}\!\Rightarrow} \mathbf{typevoid}}$$

5.3 Constants

The Core ELLA definition of constants is

const	::=	STRING [size] const1
	|	[size] const
	|	const1
const1	::=	enumerated
	|	altname & const1
	|	($\text{const}_1, \cdots, \text{const}_k$)
	|	? type
	|	()

with their transformation rules being

$$\boxed{\text{C1}}\ \frac{\begin{array}{c}[\![\textit{const1}]\!] \mathrel{=\!\boxed{\mathrm{C}}\!\Rightarrow} \textit{kconst}: \textit{ktype} \\ \textit{Get-Type}(\textit{ktype}) = \ \mathbf{typeno}(\textit{ktypeno}) \\ \textit{Is-Char}\ (\textit{ktypeno})\end{array}}{\begin{array}{c}[\![\textit{STRING}[\textit{size}]\textit{const1}]\!] \mathrel{=\!\boxed{\mathrm{C}}\!\Rightarrow} \\ \mathbf{conststring}(\textit{size}, \textit{kconst}):\ \mathbf{stringtype}(\textit{size},\ \mathbf{typeno}(\textit{ktypeno}))\end{array}}$$

$$\boxed{\text{C2}}\ \frac{[\![\textit{const}]\!] \mathrel{=\!\boxed{\mathrm{C}}\!\Rightarrow} \textit{kconst}: \textit{ktype}}{[\![[\textit{size}]\textit{const}]\!] \mathrel{=\!\boxed{\mathrm{C}}\!\Rightarrow} \mathbf{consts}([\textit{kconst}^{\textit{size}}]):\ \mathbf{types}([\textit{ktype}^{\textit{size}}])}$$

$$\boxed{C3}\ \frac{\textit{Find-Alt}(name) = \textbf{enum}(ktypeno, ktagno)}{[\![name]\!] \Rightarrow\!\boxed{C}\!\Rightarrow\ \textbf{enum}(ktypeno, ktagno)\colon\ \textbf{typeno}(ktypeno)}$$

$$\boxed{C4}\ \frac{[\![enumerated]\!] \Rightarrow\!\boxed{EM}\!\Rightarrow kenum\colon ktype}{[\![enumerated]\!] \Rightarrow\!\boxed{C}\!\Rightarrow kenum\colon ktype}$$

$$\boxed{C5}\ \frac{\begin{array}{c}[\![const1]\!] \Rightarrow\!\boxed{C}\!\Rightarrow kconst\colon ktype_1 \\ \textit{Find-Assoc}\ (altname)\ = ktype_2 \\ ktype_1 \Rightarrow\!\boxed{T}\!\Leftarrow ktype_2 \qquad \textit{Find-Alt}\ (altname)\ =\ \textbf{enum}(ktypeno, index)\end{array}}{\begin{array}{c}[\![altname\&const1]\!] \Rightarrow\!\boxed{C}\!\Rightarrow \\ \textbf{constassoc}(\ \textbf{enum}(ktypeno, index), kconst)\colon\ \textbf{typeno}(ktypeno)\end{array}}$$

$$\boxed{C6}\ \frac{\forall i \in \{1..k\} \cdot ([\![const_i]\!] \Rightarrow\!\boxed{C}\!\Rightarrow kconst_i\colon ktype_i)}{\begin{array}{c}[\![(const_1, \cdots, const_k)]\!] \Rightarrow\!\boxed{C}\!\Rightarrow \\ \textit{ConstTuple}([kconst_1, \cdots, kconst_k])\colon \textit{TypeTuple}([ktype_1, \cdots, ktype_k])\end{array}}$$

$$\boxed{C7}\ \frac{[\![type]\!] \Rightarrow\!\boxed{T}\!\Rightarrow ktype}{[\![?type]\!] \Rightarrow\!\boxed{C}\!\Rightarrow\ \textbf{constquery}(ktype)\colon ktype}$$

$$\boxed{C8}\ \frac{}{[\![()]\!] \Rightarrow\!\boxed{C}\!\Rightarrow\ \textbf{constvoid}\colon\ \textbf{typevoid}}$$

5.4 Constant Sets

Constant sets are given by

constset	::=	constset1_1 | ⋯ | constset1_k
constset1	::=	STRING [size] constset2
	|	[size] constset1
	|	constset2
constset2	::=	enumerated
	|	altname & constset2
	|	(constset_1, ⋯, constset_k)
	|	type

with transformations on them given by

$$\boxed{\mathrm{CS1}}\ \frac{\begin{array}{c}\forall i \in \{1..k\} \cdot (\llbracket \mathit{constset}_i \rrbracket =\!\boxed{\mathrm{CS}}\!\Rightarrow \mathit{kcset}_i\colon \mathit{ktype}_i) \\ \forall i \in \{1..k\} \cdot (\mathit{ktype}_i =\!\boxed{\mathrm{T}}\!= \mathit{ktype}_1)\end{array}}{\llbracket \mathit{constset}_1 \mid \cdots \mid \mathit{constset}_k \rrbracket =\!\boxed{\mathrm{CS}}\!\Rightarrow \ \mathbf{constsetalts}([\mathit{kcset}_1, \cdots, \mathit{kcset}_k])\colon \mathit{ktype}_1}$$

$$\boxed{\mathrm{CS2}}\ \frac{\begin{array}{c}\llbracket \mathit{constset2} \rrbracket =\!\boxed{\mathrm{CS}}\!\Rightarrow \mathit{kcset}\colon \mathit{ktype} \\ \mathit{Get\text{-}Type}(\mathit{ktype}) = \mathbf{typeno}(\mathit{ktypeno}) \\ \mathit{Is\text{-}Char}\ (\mathit{ktypeno})\end{array}}{\begin{array}{c}\llbracket \mathit{STRING}[\mathit{size}]\mathit{constset2} \rrbracket =\!\boxed{\mathrm{CS}}\!\Rightarrow \\ \mathbf{constsetstring}(\mathit{size}, \mathit{kcset})\colon \mathbf{stringtype}(\mathit{size},\ \mathbf{typeno}(\mathit{ktypeno}))\end{array}}$$

$$\boxed{\mathrm{CS3}}\ \frac{\llbracket \mathit{constset1} \rrbracket =\!\boxed{\mathrm{CS}}\!\Rightarrow \mathit{kcset}\colon \mathit{ktype}}{\llbracket [\mathit{size}]\mathit{constset1} \rrbracket =\!\boxed{\mathrm{CS}}\!\Rightarrow \ \mathbf{constsets}([\mathit{kcset}^{\mathit{size}}])\colon \mathbf{types}([\mathit{ktype}^{\mathit{size}}])}$$

$$\boxed{\mathrm{CS4}}\ \frac{\mathit{Find\text{-}Type\text{-}or\text{-}Alt}(\mathit{name}) = \mathit{res}}{\begin{array}{c}\llbracket \mathit{name} \rrbracket =\!\boxed{\mathrm{CS}}\!\Rightarrow \\ \mathbf{if}\ \mathit{res} = \ \mathbf{enum}(\mathit{ktypeno}, \_)\ \mathbf{then}\ \mathit{res} : \mathbf{typeno}(\mathit{ktypeno})\ \mathbf{else}\ \ \mathbf{constsetany}(\mathit{res})\colon \mathit{res}\end{array}}$$

$$\boxed{\mathrm{CS5}}\ \frac{\llbracket \mathit{enumerated} \rrbracket =\!\boxed{\mathrm{EM}}\!\rightharpoondown \mathit{kenum}\colon \mathit{ktype}}{\llbracket \mathit{enumerated} \rrbracket =\!\boxed{\mathrm{CS}}\!\Rightarrow \mathit{kenum}\colon \mathit{ktype}}$$

$$\boxed{\mathrm{CS6}}\ \frac{\begin{array}{cc}\llbracket \mathit{constset2} \rrbracket =\!\boxed{\mathrm{CS}}\!\Rightarrow \mathit{kcset}\colon \mathit{ktype}_1 & \mathit{Find\text{-}Assoc}\ (\mathit{altname}) = \mathit{ktype}_2 \\ \mathit{ktype}_1 =\!\boxed{\mathrm{T}}\!= \mathit{ktype}_2 & \mathit{Find\text{-}Alt}\ (\mathit{altname}) = \mathbf{enum}(\mathit{ktypeno}, \mathit{tagno})\end{array}}{\begin{array}{c}\llbracket \mathit{altname}\&\mathit{constset2} \rrbracket =\!\boxed{\mathrm{CS}}\!\Rightarrow \\ \mathbf{constsetassoc}(\ \mathbf{enum}(\mathit{ktypeno}, \mathit{tagno})\ , \mathit{kcset})\colon\ \mathbf{typeno}(\mathit{ktypeno})\end{array}}$$

$$\boxed{\mathrm{CS7}}\ \frac{\forall i \in \{1..k\} \cdot (\llbracket \mathit{constset}_i \rrbracket =\!\boxed{\mathrm{CS}}\!\Rightarrow \mathit{kcset}_i\colon \mathit{ktype}_i)}{\begin{array}{c}\llbracket (\mathit{constset}_1, \cdots, \mathit{constset}_k) \rrbracket =\!\boxed{\mathrm{CS}}\!\Rightarrow \\ \mathit{ConstsetTuple}([\mathit{kcset}_1, \cdots, \mathit{kcset}_k])\colon \mathit{TypeTuple}([\mathit{ktype}_1, \cdots, \mathit{ktype}_k])\end{array}}$$

5.5 Units

The complete Core ELLA unit syntax is given by

unit	::=	unit CONC unit1
	|	unit1

unit1	::=	STRING [size] unit1
	|	[size] unit1
	|	fnname unit1
	|	altname & unit1
	|	unit2 // altname
	|	unit2

unit2	::=	signalname
	|	enumerated
	|	unit2 [index]
	|	unit2 [index_{lwb} .. index_{upb}]
	|	unit2 [[unit]]
	|	REPLACE (unit, unit, unit)
	|	? type
	|	closedclause

with the transformations defined by

$$\boxed{\text{U1}}\ \frac{\mathit{Find\text{-}Sig\text{-}or\text{-}Alt}(name) = res}{[\![name]\!] \dashv\boxed{\text{U}}\Rightarrow \text{ if } res = \textbf{enum}(ktypeno, \_) \textbf{ then } res : \textbf{typeno}(ktypeno) \textbf{ else } res}$$

$$\boxed{\text{U2}}\ \frac{[\![enumerated]\!] \dashv\boxed{\text{EM}}\Rightarrow kenum: ktypeno}{[\![enumerated]\!] \dashv\boxed{\text{U}}\Rightarrow kenum: ktypeno}$$

$$\boxed{\text{U3}}\ \frac{\begin{array}{c}[\![unit]\!] \dashv\boxed{\text{U}}\Rightarrow kunit_1: ktype_1 \\ [\![unit1]\!] \dashv\boxed{\text{U}}\Rightarrow kunit_2: ktype_2 \\ ktype_{out} = Conc(ktype_1,\ ktype_2)\end{array}}{[\![unit\ CONC\ unit1]\!] \dashv\boxed{\text{U}}\Rightarrow \textbf{conc}(kunit_1, kunit_2, ktype_{out}): ktype_{out}}$$

$$\boxed{\text{U4}}\ \frac{\begin{array}{c}[\![unit1]\!] \dashv\boxed{\text{U}}\Rightarrow kunit: ktype \\ Get\text{-}Type(ktype) = \textbf{typeno}(ktypeno) \\ Is\text{-}Char\ (ktypeno)\end{array}}{\begin{array}{c}[\![STRING[size]unit1]\!] \dashv\boxed{\text{U}}\Rightarrow \\ \textbf{unitstring}(size, kunit):\ \textbf{stringtype}(size,\ \textbf{typeno}(ktypeno))\end{array}}$$

$$\boxed{\text{U5}}\ \frac{[\![unit1]\!] \dashv\boxed{\text{U}}\Rightarrow kunit: ktype}{[\![[size]unit1]\!] \dashv\boxed{\text{U}}\Rightarrow \textbf{units}([kunit^{size}]):\ \textbf{types}([ktype^{size}])}$$

$$\boxed{\text{U6}}\ \frac{\begin{array}{c}[\![unit1]\!] \dashv\boxed{\text{U}}\Rightarrow kunit: ktype \\ fnno = Find\text{-}Fn\ (fnname) \\ (\ \textbf{Env}.fndec)[fnno].inputtype \dashv\boxed{\text{T}}\models ktype\end{array}}{[\![fnname\ unit1]\!] \dashv\boxed{\text{U}}\Rightarrow \textbf{instance}(fnno, kunit):\ ((\ \textbf{Env}.fndec)[fnno].outputtype)}$$

$$\boxed{\text{U7}}\ \frac{\begin{array}{c}[\![\mathit{unit1}]\!] =\!\boxed{\text{U}}\!\Rightarrow \mathit{kunit}_1 : \mathit{ktype}_1 \\ \mathit{Find\text{-}Assoc}\ (\mathit{altname}) = \mathit{ktype}_2 \\ \mathit{ktype}_1 =\!\boxed{\text{T}}\!= \mathit{ktype}_2 \qquad \mathit{Find\text{-}Alt}\ (\mathit{altname}) = \mathbf{enum}(\mathit{ktypeno}, \mathit{tagno})\end{array}}{\begin{array}{c}[\![\mathit{altname}\&\mathit{unit1}]\!] =\!\boxed{\text{U}}\!\Rightarrow \\ \mathbf{unitassoc}(\ \mathbf{enum}(\mathit{ktypeno}, \mathit{tagno}),\ \mathit{kunit}_1):\ \ \mathbf{typeno}(\mathit{ktypeno})\end{array}}$$

$$\boxed{\text{U8}}\ \frac{\begin{array}{c}[\![\mathit{unit2}]\!] =\!\boxed{\text{U}}\!\Rightarrow \mathit{kunit}: \mathit{ktype} \qquad \mathit{Find\text{-}Assoc}\ (\mathit{altname}) = \mathit{ktypeout}, \\ \mathit{Find\text{-}Alt}\ (\mathit{altname}) = \mathbf{enum}(\mathit{typeno}, \mathit{index}), \\ \mathbf{typeno}(\mathit{typeno}) =\!\boxed{\text{T}}\!= \mathit{ktype}\end{array}}{[\![\mathit{unit2//altname}]\!] =\!\boxed{\text{U}}\!\Rightarrow\ \mathbf{extract}(\mathit{kunit},\ \mathbf{enum}(\mathit{typeno}, \mathit{index})):\ \mathit{ktypeout}}$$

$$\boxed{\text{U9}}\ \frac{\begin{array}{c}[\![\mathit{unit2}]\!] =\!\boxed{\text{U}}\!\Rightarrow \mathit{kunit}:\ \mathit{ktype} \\ \mathit{Get\text{-}Index}(\mathit{ktype}, \mathit{index}) = t\end{array}}{[\![\mathit{unit2}[\mathit{index}]]\!] =\!\boxed{\text{U}}\!\Rightarrow\ \mathbf{index}(\mathit{kunit}, \mathit{index}, t):\ t}$$

$$\boxed{\text{U10}}\ \frac{\begin{array}{c}[\![\mathit{unit2}]\!] =\!\boxed{\text{U}}\!\Rightarrow \mathit{kunit}:\ \mathit{ktype} \\ \mathit{Trim}(\mathit{ktype}, \mathit{index}_{lwb}, \mathit{index}_{upb}) = t\end{array}}{[\![\mathit{unit2}[\mathit{index}_{lwb}..\mathit{index}_{upb}]]\!] =\!\boxed{\text{U}}\!\Rightarrow\ \mathbf{trim}(\mathit{kunit}, \mathit{index}_{lwb}, \mathit{index}_{upb}, t):\ t}$$

$$\boxed{\text{U11}}\ \frac{\begin{array}{c}[\![\mathit{unit2}]\!] =\!\boxed{\text{U}}\!\Rightarrow \mathit{kunit}_1:\ \mathit{ktype}_1 \\ [\![\mathit{unit}]\!] =\!\boxed{\text{U}}\!\Rightarrow \mathit{kunit}_2:\ \mathit{ktype}_2 \\ \mathit{Find\text{-}Integer\text{-}Type}(\mathit{ktype}_2) = \mathit{ktypeno},\ l,\ u \\ \mathit{Find\text{-}Row}(\mathit{ktype}_1) = \mathit{size},\ \mathit{tt} \qquad 1 \le l \le u \le \mathit{size}\end{array}}{[\![\mathit{unit2}[[\mathit{unit}]]]\!] =\!\boxed{\text{U}}\!\Rightarrow\ \mathbf{dyindex}(\mathit{kunit}_1, \mathit{kunit}_2, \mathit{tt}):\ \mathit{tt}}$$

$$\boxed{\text{U12}}\ \frac{\begin{array}{c}[\![\mathit{unit}_1]\!] =\!\boxed{\text{U}}\!\Rightarrow \mathit{kunit}_1:\ \mathit{ktype}_1 \\ [\![\mathit{unit}_2]\!] =\!\boxed{\text{U}}\!\Rightarrow \mathit{kunit}_2:\ \mathit{ktype}_2 \\ [\![\mathit{unit}_3]\!] =\!\boxed{\text{U}}\!\Rightarrow \mathit{kunit}_3:\ \mathit{ktype}_3 \\ \mathit{Find\text{-}Integer\text{-}Type}(\mathit{ktype}_2) = \mathit{ktypeno},\ l,\ u \\ \mathit{Find\text{-}Row}(\mathit{ktype}_1) = \mathit{size},\ t \\ 1 \le l \le u \le \mathit{size} \qquad \mathit{ktype}_3 =\!\boxed{\text{T}}\!= t\end{array}}{[\![\mathit{REPLACE}(\mathit{unit}_1, \mathit{unit}_2, \mathit{unit}_3)]\!] =\!\boxed{\text{U}}\!\Rightarrow\ \mathbf{replace}(\mathit{kunit}_1, \mathit{kunit}_2, \mathit{kunit}_3):\ \mathit{ktype}_1}$$

$$\boxed{\text{U13}}\ \frac{[\![\mathit{type}]\!] =\!\boxed{\text{T}}\!\Rightarrow \mathit{ktype}}{[\![?\mathit{type}]\!] =\!\boxed{\text{U}}\!\Rightarrow\ \mathbf{unitquery}(\mathit{ktype}):\ \mathit{ktype}}$$

$$\boxed{\text{U14}}\ \frac{[\![\mathit{closedclause}]\!] =\!\boxed{\text{CC}}\!\Rightarrow \mathit{kunit}:\ \mathit{ktype}}{[\![\mathit{closedclause}]\!] =\!\boxed{\text{U}}\!\Rightarrow \mathit{kunit}:\ \mathit{ktype}}$$

5.6 Closedclause

Closed clauses are given by

closedclause ::= CASE unit OF cases ELSE unit ESAC
| ($\text{unit}_1, \cdots, \text{unit}_k$)
| BEGIN $\text{step}_1 \cdots \text{step}_{k\text{-}1}$ OUTPUT unit END
| ()

cases ::= $\text{constset}_1 : \text{unit}_1, \cdots, \text{constset}_k : \text{unit}_k$

step ::= typedec
| fndec
| LET signalname = unit .
| MAKE fnname : signalname .
| JOIN unit → signalname .

with the transformations given by

$$\boxed{\mathrm{CA}}\ \frac{[\![\mathit{constset}]\!] =\!\boxed{\mathrm{CS}}\!\Rightarrow \mathit{kconstset}\colon \mathit{ktypeconst} \qquad [\![\mathit{unit}]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow \mathit{kunit}\colon \mathit{ktype}}{[\![\mathit{constset}\colon \mathit{unit}]\!] =\!\boxed{\mathrm{CA}}\!\Rightarrow \mathit{kconstset}\colon \mathit{ktypeconst},\ \mathit{kunit}\colon \mathit{ktype}}$$

$$\boxed{\mathrm{CC1}}\ \frac{\begin{array}{c}[\![\mathit{unit}_1]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow \mathit{kunit}_{in}\colon \mathit{ktype}_{in} \\ \forall i \in \{1..k\} \cdot ([\![\mathit{case}_i]\!] =\!\boxed{\mathrm{CA}}\!\Rightarrow \mathit{kcs}_i\colon \mathit{ktypec}_i,\ \mathit{ku}_i\colon \mathit{ktypeu}_i) \\ [\![\mathit{unit}_2]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow \mathit{kunit}_{out}\colon \mathit{ktype}_{out}, \\ \forall i \in \{1..k\} \cdot (\mathit{ktypec}_i =\!\boxed{\mathrm{T}}\!= \mathit{ktype}_{in}) \qquad \forall i \in \{1..k\} \cdot (\mathit{ktypeu}_i =\!\boxed{\mathrm{T}}\!= \mathit{ktype}_{out}) \\ \forall i, j \in \{1..k\}\ i \neq j \cdot \mathit{Disjoint}(\mathit{kcs}_i,\ \mathit{kcs}_j)\end{array}}{\begin{array}{c}[\![\mathit{CASE}\ \mathit{unit}_1 \mathit{OF}\ \mathit{case}_1, \cdots, \mathit{case}_k \mathit{ELSE}\ \mathit{unit}_2 \mathit{ESAC}]\!] =\!\boxed{\mathrm{CC}}\!\Rightarrow \\ \mathbf{caseclause}(\mathit{kunit}_{in}, [\mathbf{case}(\mathit{kcs}_1, \mathit{ku}_1), \cdots, \mathbf{case}(\mathit{kcs}_k, \mathit{ku}_k)], \mathit{kunit}_{out})\colon \mathit{ktype}_{out}\end{array}}$$

$$\boxed{\mathrm{CC2}}\ \frac{\forall i \in \{1..k\} \cdot ([\![\mathit{unit}_i]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow \mathit{kunit}_i\colon \mathit{ktype}_i)}{\begin{array}{c}[\![(\mathit{unit}_1, \cdots, \mathit{unit}_k)]\!] =\!\boxed{\mathrm{CC}}\!\Rightarrow \\ \mathit{UnitTuple}([\mathit{kunit}_1, \cdots, \mathit{kunit}_k])\colon \mathit{TypeTuple}([\mathit{ktype}_1, \cdots, \mathit{ktype}_k])\end{array}}$$

$$\boxed{\mathrm{CC3}}\ \frac{\begin{array}{c}\mathit{Scope\text{-}Begin} \\ \forall i \in \{1..k\text{-}1\} \cdot ([\![\mathit{step}_i]\!] =\!\boxed{\mathrm{SP}}\!\Rightarrow \mathbf{true}) \\ [\![\mathit{unit}]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow \mathit{kunit}\colon \mathit{ktype} \\ \mathit{Not\text{-}Local\text{-}Type}(\mathit{ktype}) \qquad \mathit{Check\text{-}Joins} \qquad \mathit{Scope\text{-}End}\end{array}}{[\![\mathit{BEGIN}\ \mathit{step}_1 \cdots \mathit{step}_{k\text{-}1} \mathit{OUTPUT}\ \mathit{unit}\ \mathit{END}]\!] =\!\boxed{\mathrm{CC}}\!\Rightarrow \mathit{kunit}\colon \mathit{ktype}}$$

$$\boxed{\mathrm{CC4}}\ \frac{}{[\![\ ()\ ]\!] =\!\boxed{\mathrm{CC}}\!\Rightarrow \mathbf{unitvoid}\colon\ \mathbf{typevoid}}$$

$$\boxed{\mathsf{SP1}}\ \frac{\textit{Scope-Fn-Begin} \qquad [\textit{typedec}] \dashv\!\boxed{\mathrm{TD}}\!\Rightarrow \mathsf{true} \qquad \textit{Scope-Fn-End}}{[\textit{typedec}] \dashv\!\boxed{\mathrm{SP}}\!\Rightarrow \mathsf{true}}$$

$$\boxed{\mathsf{SP2}}\ \frac{\textit{Scope-Fn-Begin} \qquad [\textit{fndec}] \dashv\!\boxed{\mathrm{FD}}\!\Rightarrow \mathsf{true} \qquad \textit{Scope-Fn-End}}{[\textit{fndec}] \dashv\!\boxed{\mathrm{SP}}\!\Rightarrow \mathsf{true}}$$

$$\boxed{\mathsf{SP3}}\ \frac{[\textit{unit}] \dashv\!\boxed{\mathrm{U}}\!\Rightarrow \textit{kunit}\colon\ \textit{ktype}}{\begin{array}{c}[\textit{LET signalname} = \textit{unit}.] \dashv\!\boxed{\mathrm{SP}}\!\Rightarrow \\ \textit{Add-Signal}\ (\ \mathbf{signaldec}(\textit{signalname}, \textit{ktype}, \textit{kunit}),\ \mathbf{joined})\end{array}}$$

$$\boxed{\mathsf{SP4}}\ \frac{\begin{array}{c}\textit{Find-Fn}\ (\textit{fnname})\ = \textit{fnno} \\ \textit{ktype}_{in} = (\ \mathbf{Env}.\textit{fndec})[\textit{fnno}].\textit{inputtype} \\ \textit{ktype}_{out} = (\ \mathbf{Env}.\textit{fndec})[\textit{fnno}].\textit{outputtype}\end{array}}{\begin{array}{c}[\textit{MAKE fnname}\colon \textit{signalname}.] \dashv\!\boxed{\mathrm{SP}}\!\Rightarrow \\ \textit{Add-Signal}\ (\ \mathbf{signaldec}(\textit{signalname}, \textit{ktype}_{out},\ \mathbf{instance}(\textit{fnno},\ \mathbf{unitquery}(\textit{ktype}_{in}))),\ \mathbf{unjoined})\end{array}}$$

$$\boxed{\mathsf{SP5}}\ \frac{\begin{array}{c}[\textit{unit}] \dashv\!\boxed{\mathrm{U}}\!\Rightarrow \textit{kunit}\colon\ \textit{ktype}, \\ \textit{Find-Unjoined}\ (\textit{signalname})\ = \textit{fnno} \\ \textit{Find-Signal}\ (\textit{signalname})\ =\ \mathbf{signal}(\textit{signalno})\colon \textit{ktype}_{out} \\ (\ \mathbf{Env}.\textit{fndec})[\textit{fnno}].\textit{inputtype} \dashv\!\boxed{\mathrm{T}}\!\vDash \textit{ktype}\end{array}}{\begin{array}{c}[\textit{JOIN unit} \longrightarrow \textit{signalname}.] \dashv\!\boxed{\mathrm{SP}}\!\Rightarrow \\ \textit{Add-Join}\ (\ \mathbf{signaldec}(\textit{signalname}, \textit{ktype}_{out},\ \mathbf{instance}(\textit{fnno}, \textit{kunit})), \textit{signalno})\end{array}}$$

5.7 Built-In Functions

Built-in-functions (function bodies) are defined to be

```
functionbody  ::=   unit
                  | REFORM
                  | BIOP biopname
                  | DELAY ( initialvalue, ambigtime, ambigvalue, delaytime)
                  | IDELAY ( initialvalue, delaytime )
                  | SAMPLE ( interval, initialvalue, skewtime )
                  | RAM ( initialvalue )
```

with the following transformations

$$\boxed{\mathsf{BI1}}\ \frac{\textit{TypeTuple}(\textit{Flatten}(\textit{ktype}_{in})) \dashv\!\boxed{\mathrm{T}}\!\vDash \textit{TypeTuple}(\textit{Flatten}(\textit{ktype}_{out}))}{[\textit{REFORM}]\ \{\textit{ktype}_{in}, \textit{ktype}_{out}\} \dashv\!\boxed{\mathrm{BI}}\!\Rightarrow\ \mathbf{reform}}$$

$$\boxed{\mathrm{BI2}}\ \frac{\textit{Find-Biop}(biopname,\ ktype_{in},\ ktype_{out}) = \mathbf{biop}(biopname)}{[\![BIOP\ biopname]\!]\ \{ktype_{in}, ktype_{out}\} \ =\!\boxed{\mathrm{BI}}\!\Rightarrow\ \mathbf{biop}(biopname)}$$

$$\boxed{\mathrm{BI3}}\ \frac{\begin{array}{c}[\![initialvalue]\!] =\!\boxed{\mathrm{C}}\!\Rightarrow kconst_i : ktype_i \\ [\![ambigvalue]\!] =\!\boxed{\mathrm{C}}\!\Rightarrow kconst_a : ktype_a \\ ktype_{in} =\!\boxed{\mathrm{T}}\!= ktype_{out} =\!\boxed{\mathrm{T}}\!= ktype_i =\!\boxed{\mathrm{T}}\!= ktype_a \\ 0 \leq ambigtime \leq delaytime \qquad 0 < delaytime\end{array}}{\begin{array}{c}[\![DELAY(initialvalue, ambigtime, ambigvalue, delaytime)]\!]\ \{ktype_{in}, ktype_{out}\} =\!\boxed{\mathrm{BI}}\!\Rightarrow \\ \mathbf{delay}(kconst_i, ambigtime, kconst_a, delaytime)\end{array}}$$

$$\boxed{\mathrm{BI4}}\ \frac{[\![initialvalue]\!] =\!\boxed{\mathrm{C}}\!\Rightarrow kconst : ktype \qquad ktype =\!\boxed{\mathrm{T}}\!= ktype_{in} =\!\boxed{\mathrm{T}}\!= ktype_{out}}{\begin{array}{c}[\![IDELAY(initialvalue, delaytime)]\!]\ \{ktype_{in}, ktype_{out}\} =\!\boxed{\mathrm{BI}}\!\Rightarrow \\ \mathbf{idelay}(kconst, delaytime)\end{array}}$$

$$\boxed{\mathrm{BI5}}\ \frac{\begin{array}{c}[\![initialvalue]\!] =\!\boxed{\mathrm{C}}\!\Rightarrow kconst : ktype \\ \text{-}interval \leq skew \leq interval \qquad ktype_{in} =\!\boxed{\mathrm{T}}\!= ktype_{out} =\!\boxed{\mathrm{T}}\!= ktype\end{array}}{\begin{array}{c}[\![SAMPLE(interval, initialvalue, skew)]\!]\ \{ktype_{in}, ktype_{out}\} =\!\boxed{\mathrm{BI}}\!\Rightarrow \\ \mathbf{sample}(interval, kconst, skew)\end{array}}$$

$$\boxed{\mathrm{BI6}}\ \frac{\begin{array}{c}[\![initialvalue]\!] =\!\boxed{\mathrm{C}}\!\Rightarrow kconst_I : ktype_I \\ ktype_{in} = \mathbf{types}([ktype_{data}, ktype_{writeaddress}, ktype_{readaddress}, ktype_{writeenable}]) \\ ktype_{data} =\!\boxed{\mathrm{T}}\!= ktype_{out} =\!\boxed{\mathrm{T}}\!= ktype_I \\ \textit{Find-Integer-Type}\ (ktype_{writeaddress}) = -,\ lb,\ ub \\ \textit{Find-Integer-Type}\ (ktype_{readaddress}) = -,\ lb,\ ub \qquad lb = 1 \\ \textit{Check-Two-Val}\ (\textit{Get-Type}(ktype_{writeenable}))\end{array}}{[\![RAM(initialvalue)]\!]\ \{ktype_{in}, ktype_{out}\} =\!\boxed{\mathrm{BI}}\!\Rightarrow\ \mathbf{ram}(kconst_I)}$$

5.8 Type Declarations

Type declarations are defined as

typedec	::=	TYPE typename = typeornew.
typeornew	::=	type
	|	new
new	::=	NEW tagname / (lwb .. upb)
	|	NEW (typealt_1 | ⋯ | typealt_k)
	|	NEW tagname ($'\text{char}_1$ | ⋯ | $'\text{char}_k$)
typealt	::=	altname & type
	|	altname

with transformations on them by

$$\boxed{\mathsf{TD1}}\ \frac{[\mathit{new}] \dashv\boxed{\mathrm{NW}}\Rightarrow \mathit{knew}}{[\mathit{TYPE\ typename} = \mathit{new}.\,] \dashv\boxed{\mathrm{TD}}\Rightarrow \mathit{Add\text{-}Type}\ (\ \mathbf{typedec}(\mathit{typename}, \mathit{knew}))}$$

$$\boxed{\mathsf{TD2}}\ \frac{[\mathit{type}] \dashv\boxed{\mathrm{T}}\Rightarrow \mathit{ktype}}{[\mathit{TYPE\ typename} = \mathit{type}.\,] \dashv\boxed{\mathrm{TD}}\Rightarrow \mathit{Add\text{-}Type\text{-}Name}\ (\mathit{typename}, \mathit{ktype})}$$

$$\boxed{\mathsf{NW1}}\ \frac{\mathit{lwb} \leq \mathit{upb} \qquad \mathit{Add\text{-}Tag}\ (\mathit{tagname})}{[\mathit{NEW\ tagname}/(\mathit{lwb}..\mathit{upb})] \dashv\boxed{\mathrm{NW}}\Rightarrow\ \mathbf{ellaint}(\mathit{tagname}, \mathit{lwb}, \mathit{upb})}$$

$$\boxed{\mathsf{NW2}}\ \frac{\begin{array}{c}\forall i \in \{1..k\} \cdot ([\mathit{typealt}_i] \dashv\boxed{\mathrm{TA}}\Rightarrow t_i) \\ \forall i, j \in \{1..k\},\ i \neq j \cdot (t_i.\mathit{tagname} \neq t_j.\mathit{tagname})\end{array}}{[\mathit{NEW}\ (\mathit{typealt}_1 \mid \cdots \mid \mathit{typealt}_k)] \dashv\boxed{\mathrm{NW}}\Rightarrow\ \mathbf{tags}([t_1, \cdots, t_k])}$$

$$\boxed{\mathsf{NW3}}\ \frac{\begin{array}{c}\forall i, j \in \{1..k\}\ i \neq j \cdot (\mathit{char}_i \neq \mathit{char}_j) \\ \mathit{Add\text{-}Tag}\ (\mathit{tagname})\end{array}}{\begin{array}{c}[\mathit{NEW\ tagname}('\mathit{char}_1 \mid \cdots \mid '\mathit{char}_k)] \dashv\boxed{\mathrm{NW}}\Rightarrow \\ \mathbf{chars}(\mathit{tagname}, [\mathit{char}_1, \cdots, \mathit{char}_k])\end{array}}$$

$$\boxed{\mathsf{TA1}}\ \frac{[\mathit{type}] \dashv\boxed{\mathrm{T}}\Rightarrow \mathit{ktype} \qquad \mathit{Add\text{-}Tag}\ (\mathit{altname})}{[\mathit{altname}\&\mathit{type}] \dashv\boxed{\mathrm{TA}}\Rightarrow\ \mathbf{tag}(\mathit{altname}, \mathit{ktype})}$$

$$\boxed{\mathsf{TA2}}\ \frac{\mathit{Add\text{-}Tag}\ (\mathit{altname})}{[\mathit{altname}] \dashv\boxed{\mathrm{TA}}\Rightarrow\ \mathbf{tag}(\mathit{altname}, \{\mathsf{nil}\})}$$

5.9 Function Declarations

Function declarations are given by

fndec	::=	FN fnname = input → type : functionbody.
input	::=	($\text{terminal}_1, \cdots, \text{terminal}_k$)
	|	()
terminal	::=	type : signalname
	|	type

and the transformations on them by

$$\boxed{\mathsf{FD1}}\ \frac{\begin{array}{c}[\mathit{input}] =\!\boxed{\mathrm{IN}}\!\Rightarrow \mathit{ktype}_{\mathit{inputs}}\\ [\mathit{type}] =\!\boxed{\mathrm{T}}\!\Rightarrow \mathit{ktype}_{\mathit{out}}\\ [\mathit{unit}] =\!\boxed{\mathrm{U}}\!\Rightarrow \mathit{kunit}\colon \mathit{ktype}\\ \mathit{ktype}_{\mathit{out}} =\!\boxed{\mathrm{T}}\!= \mathit{ktype}\end{array}}{\begin{array}{c}[\mathit{FN}\ \mathit{fnname} = \mathit{input} \longrightarrow \mathit{type}\colon \mathit{unit}.\,] =\!\boxed{\mathrm{FD}}\!\Rightarrow\\ \mathit{Add\text{-}Fn}\ (\ \mathbf{fndec}(\mathit{fnname}, \mathit{ktype}_{\mathit{inputs}},\ \mathbf{Env}.\mathit{sigdec}, \mathit{ktype}_{\mathit{out}}, \mathit{kunit}))\end{array}}$$

$$\boxed{\mathsf{FD2}}\ \frac{\begin{array}{c}[\mathit{input}] =\!\boxed{\mathrm{IN}}\!\Rightarrow \mathit{ktype}_{\mathit{inputs}}\\ [\mathit{type}] =\!\boxed{\mathrm{T}}\!\Rightarrow \mathit{ktype}_{\mathit{out}}\\ [\mathit{builtin}]\ \{\mathit{Get\text{-}Type}(\mathit{ktype}_{\mathit{inputs}}), \mathit{Get\text{-}Type}(\mathit{ktype}_{\mathit{out}})\} =\!\boxed{\mathrm{BI}}\!\Rightarrow \mathit{kbuiltin}\end{array}}{\begin{array}{c}[\mathit{FN}\ \mathit{fnname} = \mathit{input} \longrightarrow \mathit{type}\colon \mathit{builtin}.\,] =\!\boxed{\mathrm{FD}}\!\Rightarrow\\ \mathit{Add\text{-}Fn}\ (\ \mathbf{fndec}(\mathit{fnname}, \mathit{ktype}_{\mathit{inputs}},\ \mathbf{Env}.\mathit{sigdec}, \mathit{ktype}_{\mathit{out}}, \mathit{kbuiltin}))\end{array}}$$

$$\boxed{\mathsf{IN1}}\ \frac{\forall i \in \{1..k\} \cdot ([\mathit{terminal}_i] =\!\boxed{\mathrm{TM}}\!\Rightarrow \mathit{ktype}_i)}{[(\mathit{terminal}_1, \cdots, \mathit{terminal}_k)] =\!\boxed{\mathrm{IN}}\!\Rightarrow \mathit{TypeTuple}([\mathit{ktype}_1, \cdots, \mathit{ktypes}_k])}$$

$$\boxed{\mathsf{IN2}}\ \frac{}{[\ ()\ ] =\!\boxed{\mathrm{IN}}\!\Rightarrow \ \mathbf{typevoid}}$$

$$\boxed{\mathsf{TM1}}\ \frac{\begin{array}{c}[\mathit{type}] =\!\boxed{\mathrm{T}}\!\Rightarrow \mathit{ktype}\\ \mathit{Add\text{-}Signal}(\ \mathbf{signaldec}(\mathit{signalname}, \mathit{ktype}, \ \mathbf{input}), \ \mathbf{joined})\end{array}}{[\mathit{type}\colon \mathit{signalname}] =\!\boxed{\mathrm{TM}}\!\Rightarrow \mathit{ktype}}$$

$$\boxed{\mathsf{TM2}}\ \frac{[\mathit{type}] =\!\boxed{\mathrm{T}}\!\Rightarrow \mathit{ktype}}{[\mathit{type}] =\!\boxed{\mathrm{TM}}\!\Rightarrow \mathit{ktype}}$$

5.10 Closure

A Closure is defined to be

declaration	::=	typedec
	|	fndec
closure	::=	$\text{declaration}_1 \cdots \text{declaration}_k$

with the following transforms

$$\boxed{\mathsf{D1}}\ \frac{[\mathit{typedec}] =\!\boxed{\mathrm{TD}}\!\Rightarrow \mathsf{true}}{[\mathit{typedec}] =\!\boxed{\mathrm{D}}\!\Rightarrow \mathsf{true}}$$

$$\boxed{\text{D2}}\ \frac{\textit{Stackenv} \qquad [\![\textit{fndec}]\!] \Rightarrow\!\boxed{\text{FD}}\!\Rightarrow \text{true} \qquad \textit{Unstackenv}}{[\![\textit{fndec}]\!] \Rightarrow\!\boxed{\text{D}}\!\Rightarrow \text{true}}$$

$$\boxed{\text{CL}}\ \frac{\forall i \in \{1..k\} \cdot ([\![\textit{declaration}_i]\!] \Rightarrow\!\boxed{\text{D}}\!\Rightarrow \text{true})}{[\![\textit{declaration}_1 \cdots \textit{declaration}_k]\!] \Rightarrow\!\boxed{\text{CL}}\!\Rightarrow \text{true}}$$

$$\boxed{\text{KERNEL}}\ \frac{[\![\textit{closure}]\!] \Rightarrow\!\boxed{\text{CL}}\!\Rightarrow \text{true}}{[\![\textit{closure}]\!] \Rightarrow\!\boxed{\text{KERNEL}}\!\Rightarrow (\ \textbf{Env}.\textit{typedec},\ \textbf{Env}.\textit{fndec})}$$

6 Software Implementation

The functions and transformational rules given in the previous sections have been translated into Lisp code. This work forms parts of the commitment to the IED project "Formal Verification Support for ELLA" which is developing a Lisp environment and toolset for Formal Verification. The Lisp system and the Lexical Analyser were made available for the project by Harlequin Ltd. of Cambridge.

The **Kernel** data structures were translated into Lisp structures, for example **typeno**(typeno) became

(defstruct $typeno(typeno))

where the $ is used to prefix all **Kernel** data structures (corresponding to bold type in Appendix C). Translation of the functions defined in this document to Lisp was straightforward with only a few additional functions needed to handle recursive structures.

7 Transformation Examples

In this section we give examples of Core ELLA descriptions which have been compiled into **Kernel** data structures by means of the software implementation. The implementation has been carried out using the Harlequin Lisp system (LispWorks), with the resulting code being incorporated into the verification environment of the project.

The first example is taken from [MH91], whilst the second example contains Core ELLA text which will test each transformation rule (although not for ever possible combination of constructs).

7.1 Simple Example

This example is taken from [MH91] and is reproduced below

```
TYPE bool = NEW (t | f).

FN NOR = (bool:in1, bool:in2) -> bool:
CASE (in1, in2) OF
  (f,t):f,
  (t,f):f,
  (t,t):f
 ELSE t
ESAC.


FN A = (bool:in1, bool:in2) -> bool:
BEGIN
  LET ip = (in1,in2).
  FN B = (bool:ip1, bool:ip2) -> bool: NOR(ip1,ip2).
  MAKE B:b.
  JOIN ip -> b.
  OUTPUT b
END.
```

In [MH91] it was shown how each of the transformation rules was applied to these functions and the final **Kernel** environment was deduced. By means of the Lisp implementation of the rules, the final **Kernel** environment was calculated and is shown below, where it has been printed out by means of a specially written Lisp printer.

```
TYPEDECS>
TYPEDEC ("bool"  Tags([Tag(t, NIL),Tag(f, NIL)]))
FNDECS>
FNDEC( NOR,
      Types([Typeno(1),Typeno(1)]),
     [Signaldec("in1", Typeno(1), input),
     Signaldec("in2", Typeno(1), input)],
      Typeno(1),
     Caseclause(Units([Signal(1),Signal(2)]),
         [Case(Constsets([Enum(1, 2),Enum(1, 1)]), Enum(1, 2)),
         Case(Constsets([Enum(1, 1),Enum(1, 2)]), Enum(1, 2)),
         Case(Constsets([Enum(1, 1),Enum(1, 1)]), Enum(1, 2))],
         Enum(1, 1)))
FNDEC( B,
      Types([Typeno(1),Typeno(1)]),
     [Signaldec("ip1", Typeno(1), input),
     Signaldec("ip2", Typeno(1), input)],
      Typeno(1),
     Instance(1, Units([Signal(1),Signal(2)])))
FNDEC( A,
      Types([Typeno(1),Typeno(1)]),
     [Signaldec("in1", Typeno(1), input),
     Signaldec("in2", Typeno(1), input),
     Signaldec("ip", Types([Typeno(1),Typeno(1)]), Units([Signal(1),Signal(2)])),
     Signaldec("b", Typeno(1), Instance(2, Signal(3)))],
```

```
      Typeno(1),
     Signal(4))
SIGDECS>
FNMAPS>
LCLFNMAPS>
map("A", 3)
map("NOR", 1)
TYNAMEMAPS>
LCLTYNAMEMAPS>
map("bool", Typeno(1))
map("f", Consttag(1))
map("t", Consttag(1))
SIGNAMEMAPS>
LCLSIGNAMEMAPS>
USEDTYNAMES>
USEDFNNAMES>
USEDSIGNAMES>
```

This environment is identical to the environment given in [MH91].

Due to the nature of the **Kernel** data structures it is possible to print out the above environment in a more readable format. This format takes on a layout which is a 'recursive-LET' ELLA-like form e.g.

```
TYPEDEC bool = NEW (t | f)
FNDECS>
FNDEC NOR = ((bool, bool)) -> (bool):
     BEGIN (LET in1 = input.  LET in2 = input.)
     OUTPUT  CASE (in1,in2) OF
            ( ((f ,t ) : f )
              ((t ,f ) : f )
              ((t ,t ) : f ))
             ELSE t
             ESAC
     END.
FNDEC B = ((bool, bool)) -> (bool):
     BEGIN (LET ip1 = input.  LET ip2 = input.)
     OUTPUT NOR (ip1,ip2)
     END.
FNDEC A = ((bool, bool)) -> (bool):
     BEGIN (LET in1 = input.
            LET in2 = input.
            LET ip = (in1,in2).
            LET b = B ip.)
     OUTPUT b
     END.
SIGDECS>
FNMAPS>
LCLFNMAPS>
```

```
map("A" : 3)
map("NOR" : 1)
TYNAMEMAPS>
LCLTYNAMEMAPS>
map("bool" : bool)
map("f" : "bool")
map("t" : "bool")
SIGNAMEMAPS>
LCLSIGNAMEMAPS>
USEDTYNAMES>
USEDFNNAMES>
USEDSIGNAMES>
```

The 'map' fields show which declarations are available for use at the outermost level, hence function 'B' does not appear in either of the function maps.

7.2 Transformational Rules Test

In this section we present a test program which goes through the different transformation rules.

The first part of the test program looks at individual expressions, for example constant expressions are created in a delay expression.

```
FN FBODY_30 = (char:one) -> char: IDELAY( c'd, 3).                  #enum#

FN FBODY_31 = (string:one) -> string: IDELAY( c"abcfghde", 3).      #string#

FN FBODY_32 = (string:one) -> string: IDELAY( STRING [8] c'c, 3).  #conststring#

FN FBODY_33 = ((enum, int):one) -> (enum, int): IDELAY( (e1,i/1), 3).  #consts#

FN FBODY_34 = (assoc:one) -> assoc: IDELAY( val & i/7, 3).         #constassoc#

FN FBODY_35 = (enum:one) -> enum: IDELAY( ?enum, 3).               #constquery#
```

Scoping rules are considered in the second part of the test program where a number of functions are given for testing the different forms of enumerated values applied to units as well as testing the scoping rules. For example

```
FN NUMBER_4 = (string:one, string: two, [4]string:three, range: four)->string:
BEGIN
   TYPE senum = NEW (se1|se2|se3|se4).                              #typedec#
   TYPE srow = [4]string.
   TYPE sname = string.
   FN FBODY_1 = (sname:one, srow: two) -> (srow, sname): REFORM.
   FN INNER_1 = (string:one, sname: two, srow:three, range: four) -> string:
   BEGIN
     LET sig_1 = FBODY_1 (one, three).                               #instance#
     LET sig_2 = c"Abcdefgh".                                        #string#
     LET sig_3 = one.                                                #signal#
     LET sig_4 = three[2].                                           #index#
     LET sig_5 = three[3..4].                                        #trim#
     LET sig_6 = three[[four]].                                      #dyindex#
```

```
      LET sig_7 = ?sname.                                                  #unitquery#
      LET sig_8 = [5](FBODY_1 (two, three)).                               #units(row)#
      LET sig_9 = REPLACE(three, four, one).                               #replace#
      LET clause = BEGIN FN FBODY_5 = (string:j)->string:j.                #begin..end#
                         MAKE FBODY_5: fbody_5.
                         JOIN two -> fbody_5.
                         OUTPUT fbody_5
                   END.
      FN INNER_2 = (string:one, sname: two, srow:three, range: four) -> string:
                   one.                                                    #fndec#
      MAKE FBODY_1: fbody_1.
      JOIN (one, sig_8[1][1][1] CONC sig_8[1][1][2..4]) -> fbody_1.        #conc#
      OUTPUT sig_5[1]
    END.
    MAKE INNER_1 : inner_1.                                                #make#
    JOIN (one, two, three, four) -> inner_1.                               #join#
    LET output = inner_1.                                                  #let#
    OUTPUT output
END.
```

The complete test program is given in appendix F.

This test program has been successfully submitted to the Lisp implementation of the rules defined in this document. The complete file took 2.6 secs to translate into the **Kernel** via the implementation on a SparcStation 2 and the final environment contained 72 functions and 30 global identifiers.

7.3 From Full ELLA to the Kernel

In appendix G we present an example taken from a description in High level ELLA using sequences, down to Core ELLA via the full ELLA system and then into the Kernel via the Lisp system. The example is based on a three pump controller, the functionality of which is defined by

A reservoir is connected to a lake by a pipe line. Water is taken from the lake to the reservoir by a system of three pumps. Three level sensors are installed on the reservoir. Their outputs are denoted by signals a_1, a_2, a_3. Signal a_i is 0 when the water is above level i, for i = 1,2,3 and has a value 1 when the water is below level i. The number of pumps that are on at any one time depends on the water level in the reservoir. In particular: if the water level is between level 1 and 2, then one pump should be in operation; if the water level is between level 2 and 3, then two pumps should be in operation; if the water level is below level 3, then three pumps should be in operation. Of course, if the water level is above level 1 then no pumps should be in operation. In order to equalise wear on the pumps, they should come into operation in a cyclic manner.

Appendix G gives the descriptions for high,medium and Core ELLA as well as the Kernel description which is written out in the recursive-let format.

7.4 Microprocessor Transformations

A number of high level descriptions of microprocessors where submitted to the complete transformation system from Full ELLA to Core ELLA to the **Kernel**. Table 7.1 shows the resulting

CPU times for the Core-to-Kernel phase together with the number of lines of **Kernel** code generated. The number of lines of original high level ELLA and the resulting Core ELLA descriptions are given for comparison. The Lisp code which produced these results was compiled and loaded using the Harlequin Lisp system, this gave a very significant improvement in speed for the transformation from Core-to-Kernel over equivalent interpreted code. The translator is implemented on a SparcStation 2.

μprocessor	High Level ELLA lines	Core ELLA Lines	Kernel Lines	No. of Functions	No. of ids	Core-to-Kernel Compile Time
68000	29[illegible]8	4280	72000	197	124	93.6 secs
Viper	2234	2625	24000	229	52	25.0 secs
6800	1088	2205	5300	46	212	23.0 secs
6502	991	1623	3900	41	88	16.3 secs

Table 7.1 Microprocessor Translation Times

The number of functions and identifiers correspond to those available in the final **Kernel** environment. The translation from High level ELLA to Core ELLA was carried out by means of the software transformations in the full ELLA system.

8 Conclusions

In this document the formal definition of the Lisp implementation of the transformational mapping from Core ELLA to the **Kernel** has been given. Examples of use of the Lisp implementation have been presented.

9 Acknowledgements

The authors would like to acknowledge the support of the IED project 4/1/1357 "Formal Verification Support for ELLA". The authors would also like to thank Harlequin Ltd. of Cambridge for making Lispworks and their Lexical analyser available for this work.

A Glossary of Symbols

Functions

$f: D_1 \times D_2 \to R$	signature
$f \underline{\Delta} \cdots$	function definition
$f(d)$	application
if $\cdots$ *then* $\cdots$ *else* $\cdots$	condtional
let $z = \cdots$ *in* $\cdots$	local definition
case z *of* $\cdots$ *else* $\cdots$ *end*	choice
pre	pre-condition

Composite Objects

Object::fieldname:fieldtype	Record Object definition
$\mu(E, s \mapsto t)$	change field s of E to hold t
$\mu(E, s \mapsto (E.s \dagger t))$	update field s of E by overwriting with t

Sets

T-set	finite subset of T
$\{t_1, \cdots, t_k\}$	set enumeration
$\{\}$	empty set
$t \in T$	set membership
$T_1 \cap T_2$	set intersection
$T_1 \cup T_2$	set union
$T_1 \subseteq T_2$	set containment
$\mathbb{Z}$	$\{\cdots, -1, 0, 1, \cdots\}$
$\mathbb{N}_1$	$\{1, 2, \cdots\}$
$\mathbb{B}$	{true, false}

Maps

$D \xrightarrow{m} R$	finite map
$\mathsf{dom}\ m$	domain
$\mathsf{rng}\ m$	range
$m_1 \dagger m_2$	overwriting

Sequences

S^*	finite sequence
$[s_1, \cdots, s_k]$	sequence enumeration
$[\,]$	empty sequence
$\mathsf{len}\ l$	length of sequence l
$s_1 \frown s_2$	concaternation
$\iota(i \in \mathsf{inds}\ sequence) \cdot sequence[i] = s$	The unique element of *sequence* which equals s

Transformation

Env = **env**(_,_,_,_,_,_,_,_,_,_,_,_,_,_)	Transformation Environment
Env.*fieldname*	field selection in the **Kernel**
(**Env**.*filedname*)[*number*]	indexing
inv(**Env**)	invariant of environment **Env**
[*Core-Syntax*] $=\!\boxed{\text{Rule}}\!\Rightarrow$ *Kernel-Expressions*	formal transformation
$=\!\boxed{\text{U}}\!\Rightarrow$ _:_,_	syntactic separaters (: ,)
$ktype_1$ $=\!\boxed{\text{T}}\!=$ $ktype_2$	type equality in the **Kernel**

Kernel

typedec(_,_)	**Kernel** data structure with wild-card entries
TypeOpt	Type structure with optional element nil
TypeSeq	Non-empty sequence of *Types*
kType	'Type' belonging in the **Kernel**

B Core ELLA Composite Syntax

B.1 Basic Notation

abc ∈ Abc	'abc' is an element of the set 'Abc'
b ::= c	the syntax definition of 'b' is 'c'
|	the separator of alternatives in a syntax definition
|	ELLA separator of alternatives
$d_1 \cdots d_k$	one or more occurrences of 'd'
$d_1, \cdots, d_k$	one or more occurrences of 'd' separated by ','. Note if k=1 then no ',' is present.
$d_1, \cdots, d_{k-1}$	zero or more occurrences of 'd' separated by ','. Note if k=0 then no ',' is present.
$\mathbb{Z}$	$\{ \cdots, -1, 0, 1, \cdots \}$
$\mathbb{N}_1$	$\{ 1, 2, \cdots \}$
Identifier	Lower case letter
Fnname	Upper case letter or symbol
Constant	ELLA constant expression
Character	Any printable character

B.2 Syntactic Categories

typename	∈	Identifier	(ELLA type name e.g. lower case)
signalname	∈	Identifier	(ELLA signal name)
tagname	∈	Identifier	(ELLA tagged type name)
altname	∈	Identifier	(ELLA enumerated type alternative)
fnname	∈	Fnname	(ELLA function name e.g. upper case or symbol)
biopname	∈	Fnname	(ELLA BIOP name e.g. upper case)
z	∈	$\mathbb{Z}$	(An integer)
lwb, upb	∈	$\mathbb{Z}$	(An integer)
j,k	∈	$\mathbb{N}_1$	(A non-zero positive integer)
index	∈	$\mathbb{N}_1$	(A non-zero positive integer)
size	∈	$\mathbb{N}_1$	(A non-zero positive integer)
interval	∈	$\mathbb{N}_1$	(ELLA timing interval)
ambigtime	∈	$\mathbb{N}_1$	(Ambiguity delay time)
delaytime	∈	$\mathbb{N}_1$	(delay time)
skewtime	∈	$\mathbb{N}_1$	(skew delay)
initialvalue	∈	Constant	(Delay, Retiming or Ram initialisation value)
ambigvalue	∈	Constant	(Delay ambiguity value)
char	∈	Character	(A printable character e.g. 'a')
string	∈	String	(A string of printable characters e.g. 'abc')

B.3 Syntactic Definitions

Enumerated

enumerated ::= altname
| tagname / z
| tagname 'char
| tagname "string"

Type

type ::= typename
| STRING [size] typename
| [size] type
| ($\text{type}_1, \cdots, \text{type}_k$)
| ()

Constant

const ::= STRING [size] const1
| [size] const
| const1

const1 ::= enumerated
| altname & const1
| ($\text{const}_1, \cdots, \text{const}_k$)
| ? type
| ()

Constset

constset ::= $\text{constset}_1 \mid \cdots \mid \text{constset}_k$

constset1 ::= STRING [size] constset2
| [size] constset1
| constset2

constset2 ::= enumerated
| altname & constset2
| ($\text{constset}_1, \cdots, \text{constset}_k$)
| type

Unit

```
unit          ::=    unit CONC unit1
                   | unit1

unit1         ::=    STRING [ size ] unit1
                   | [ size ] unit1
                   | fnname unit1
                   | altname & unit1
                   | unit2 // altname
                   | unit2

unit2         ::=    signalname
                   | enumerated
                   | unit2 [ index ]
                   | unit2 [ index_lwb .. index_upb ]
                   | unit2 [[ unit ]]
                   | REPLACE (unit, unit, unit)
                   | ? type
                   | closedclause
```

Closedclause

```
closedclause  ::=    CASE unit OF cases ELSE unit ESAC
                   | ( unit_1, ..., unit_k )
                   | BEGIN step_1 ... step_k-1 OUTPUT unit END
                   | ()

cases         ::=    constset_1 : unit_1, ..., constset_k : unit_k

step          ::=    typedec
                   | fndec
                   | LET signalname = unit .
                   | MAKE fnname : signalname .
                   | JOIN unit → signalname .
```

Function Body

```
functionbody  ::=    unit
                   | REFORM
                   | BIOP biopname
                   | DELAY ( initialvalue, ambigtime, ambigvalue, delaytime )
                   | IDELAY ( initialvalue, delaytime )
                   | SAMPLE ( interval, initialvalue, skewtime )
                   | RAM ( initialvalue )
```

Type Declaration

typedec	::=	TYPE typename = typeornew.
typeornew	::=	type
	|	new
new	::=	NEW tagname / (lwb .. upb)
	|	NEW (typealt_1 | ··· | typealt_k)
	|	NEW tagname ('char_1 |··· | 'char_k)
typealt	::=	altname & type
	|	altname

Function Declaration

fndec	::=	FN fnname = input → type : functionbody.
input	::=	(terminal_1, ···, terminal_k)
	|	()
terminal	::=	type : signalname
	|	type

Closure

declaration	::=	typedec
	|	fndec
closure	::=	$\text{declaration}_1 \cdots \text{declaration}_k$

C Kernel of ELLA Data Structure

C.1 Conventions

abc	$\in$	Abc (ie. it is an element of the set Abc)
Indexer, Size, Fnno	$\subseteq$	$\mathbb{N}_1$
Typeno, Tagno, Inputno	$\subseteq$	$\mathbb{N}_1$
Signalno, Delaytime	$\subseteq$	$\mathbb{N}_1$
Interval, Ambigtime	$\subseteq$	$\mathbb{N}$
Skew	$\subseteq$	$\mathbb{Z}$
Inputtype, Outputtype	$\subseteq$	Type
Initialvalue, Ambigvalue	$\subseteq$	Const
Fnname, Biopname	$\subseteq$	Upper case identifier or operator
Name, Signalname	$\subseteq$	Lower case identifier
Typename, Tagname	$\subseteq$	Lower case identifier
Lowerbound, Upperbound	$\subseteq$	positive or negative integer
Character	$\subseteq$	printable character

C.2 Data Structures

Enumerated

Enumerated	::=		Enum
		|	**string**(Typeno × TagnoSeq)
Enum	::=		**enum**(Typeno × Tagno)

Types

Type	::=		**typeno**(Typeno)
		|	**typename**(Typename × Type)
		|	**stringtype**(Size × Type)
		|	**types**(TypeSeq)
		|	**typevoid**

Constants

Const	::=		Enumerated
		|	**conststring**(Size × Const)
		|	**consts**(ConstSeq)
		|	**constassoc**(Enum × Const)
		|	**constquery**(Type)
		|	**constvoid**

Constant Sets

Constset	::=	Enumerated
	|	**constsetalts**(ConstsetSeq)
	|	**constsetstring**(Size × Constset)
	|	**constsets**(ConstsetSeq)
	|	**constsetassoc**(Enum × Constset)
	|	**constsetany**(Type)

Units

Unit	::=	Enumerated
	|	**conc**(Unit × Unit × Outputtype)
	|	**unitstring**(Size × Unit)
	|	**units**(UnitSeq)
	|	**instance**(Fnno × Unit)
	|	**unitassoc**(Enum × Unit)
	|	**extract**(Unit × Enum)
	|	**signal**(Signalno)
	|	**index**(Unit × Indexer × Outputtype)
	|	**trim**(Unit × Indexer × Indexer × Outputtype)
	|	**dyindex**(Unit × Unit × Outputtype)
	|	**replace**(Unit × Unit × Unit)
	|	**unitquery**(Type)
	|	**caseclause**(Unit × CaseSeq × Unit)
	|	**unitvoid**
Case	::=	**case**(Constset × Unit)

Function Declarations

Fndec	::=	**fndec**(Fnname × Inputtype × SignaldecSeq × Outputtype × Fnbody)
Signaldec	::=	**signaldec**(Signalname × Type × Unitorinput)
Unitorinput	::=	Unit
	|	**input**
Fnbody	::=	Unit
	|	**reform**
	|	**biop**(Biopname)
	|	**delay**(Initialvalue × Ambigtime × Ambigvalue × Delaytime)
	|	**idelay**(Initialvalue × Delaytime)
	|	**sample**(Interval × Initialvalue × Skew)
	|	**ram**(Initialvalue)

Type Declarations

Typedec	::=	**typedec**(Typename × New)
New	::=	**tags**(TagSeq)
	|	**ellaint**(Tagname × Lowerbound × Upperbound)
	|	**chars**(Tagname × CharacterSeq)
Tag	::=	**tag**(Tagname × TypeOpt)

Closures

Closure	::=	TypedecSeq × FndecSeq

Intentionally Blank

D Signatures

In this appendix we give a complete list of the signatures of the transformational functions used in this document.

SetEnv	:	(Env)$\mathbb{B}$
Stackenv	:	()$\mathbb{B}$
Unstackenv	:	()$\mathbb{B}$
Hdstack	:	()Env
Scope-Fn-Begin	:	()
Scope-Fn-End	:	()
Scope-Begin	:	()
Scope-End	:	()
Scope-End-Add-Fn	:	()$\mathbb{B}$
Local-Scope-Rule	:	Name $\longrightarrow \mathbb{B}$

Check-Joins	:	$\emptyset \longrightarrow \mathbb{B}$
Check-Two-Val	:	kType $\longrightarrow \mathbb{B}$
Check-Fn	:	Fnname $\longrightarrow \mathbb{B}$
Check-Typename	:	Name $\longrightarrow \mathbb{B}$
Check-Signal	:	Signalname $\longrightarrow \mathbb{B}$

Add-Fn	:	Fndec $\longrightarrow \mathbb{B}$
Add-Type	:	Typedec $\longrightarrow \mathbb{B}$
Add-Signal	:	Signaldec $\times$ Sort $\longrightarrow \mathbb{B}$
Add-Join	:	Signaldec $\times$ Signalno $\longrightarrow \mathbb{B}$
Add-Tag	:	Tagname $\longrightarrow \mathbb{B}$
Add-Type-Name	:	Typename $\times$ kType $\longrightarrow \mathbb{B}$

Find-Lower-Nm	:	Name $\longrightarrow$ Typetag $\cup$ Sig
Find-Type-or-Alt	:	Name $\longrightarrow$ kType $\cup$ kEnum
Find-Sig-or-Alt	:	Name $\longrightarrow$ (kUnit $\times$ kType) $\cup$ kEnum
Find-Fn	:	Fnname $\longrightarrow$ Fnno
Find-Unjoined	:	Signalname $\longrightarrow$ Fnno
Find-Type	:	Typename $\longrightarrow$ kType
Find-Alt	:	Altname $\longrightarrow$ kEnum
Find-ELLAint	:	Tagname $\longrightarrow$ Typeno $\times$ Lowerbound $\times$ Upperbound
Find-Integer-Type	:	kType $\longrightarrow$ Typeno $\times$ Lowerbound $\times$ Upperbound
Find-Char	:	Tagname $\times$ Char $\longrightarrow$ kEnum
Find-Signal	:	Signal $\longrightarrow$ kUnit $\times$ kType
Find-Assoc	:	Altname $\longrightarrow$ kType
Find-Row	:	kType $\longrightarrow \mathbb{N}_1 \times$ kType
Find-Biop	:	Biopname $\times$ kType $\times$ kType $\longrightarrow$ kFnbody

Get-Type	:	kType $\longrightarrow$ kType
Type-Equals	:	kType $\times$ kType $\longrightarrow \mathbb{B}$
Biop-Type-Equals	:	kType $\times$ kType $\longrightarrow \mathbb{B}$
Get-Index	:	kType $\times\ \mathbb{N}_1 \longrightarrow$ kType
Trim	:	kType $\times\ \mathbb{N}_1 \times \mathbb{N}_1 \longrightarrow$ kType
Conc	:	kType $\times$ kType $\longrightarrow$ kType
Flatten	:	kType $\longrightarrow$ kTypeSeq
Is-Char	:	$\mathbb{N}_1 \longrightarrow \mathbb{B}$

TypeTuple	:	$\mathbf{kTypeSeq} \longrightarrow \mathbf{kType}$
ConstTuple	:	$\mathbf{kConstSeq} \longrightarrow \mathbf{kConst}$
ConstsetTuple	:	$\mathbf{kConstsetSeq} \longrightarrow \mathbf{kConstset}$
UnitTuple	:	$\mathbf{kUnitSeq} \longrightarrow \mathbf{kUnit}$
Disjoint	:	$\mathbf{kConstset} \times \mathbf{kConstset} \longrightarrow \mathbb{B}$
Not-Local-Type	:	$\mathbf{kType} \longrightarrow \mathbb{B}$

$\_ \dashv\boxed{\mathrm{EM}}\Rightarrow \_ : \_$	$\subseteq$	$\mathit{Enum} \times \mathit{kEnumerated} \times \mathit{kType}$
$\_ \dashv\boxed{\mathrm{T}}\Rightarrow \_$	$\subseteq$	$\mathit{Type} \times \mathit{kType}$
$\_ \dashv\boxed{\mathrm{C}}\Rightarrow \_ : \_$	$\subseteq$	$\mathit{Const} \times \mathit{kConst} \times \mathit{kType}$
$\_ \dashv\boxed{\mathrm{CS}}\Rightarrow \_ : \_$	$\subseteq$	$\mathit{Constset} \times \mathit{kConstset} \times \mathit{kType}$
$\_ \dashv\boxed{\mathrm{U}}\Rightarrow \_ : \_$	$\subseteq$	$\mathit{Unit} \times \mathit{kUnit} \times \mathit{kType}$
$\_ \dashv\boxed{\mathrm{CC}}\Rightarrow \_ : \_$	$\subseteq$	$\mathit{Closedclause} \times \mathit{kUnit} \times \mathit{kType}$
$\_ \dashv\boxed{\mathrm{CA}}\Rightarrow \_ : \_ , \_ : \_$	$\subseteq$	$\mathit{Case} \times \mathit{kConstset} \times \mathit{kType} \times \mathit{kUnit} \times \mathit{kType}$
$\_ \dashv\boxed{\mathrm{SP}}\Rightarrow \_$	$\subseteq$	$\mathit{Step} \times \mathbb{B}$
$\_ \dashv\boxed{\mathrm{NW}}\Rightarrow \_$	$\subseteq$	$\mathit{New} \times \mathit{kNew}$
$\_ \dashv\boxed{\mathrm{TA}}\Rightarrow \_$	$\subseteq$	$\mathit{Typealt} \times \mathit{Tag}$
$\_ \dashv\boxed{\mathrm{TD}}\Rightarrow \_$	$\subseteq$	$\mathit{Typedec} \times \mathbb{B}$
$\_ \dashv\boxed{\mathrm{FD}}\Rightarrow \_$	$\subseteq$	$\mathit{Fndec} \times \mathbb{B}$
$\_ \{\_ , \_\} \dashv\boxed{\mathrm{BI}}\Rightarrow \_$	$\subseteq$	$\mathit{Builtin} \times \mathit{kType} \times \mathit{kType} \times \mathit{kBuiltin}$
$\_ \dashv\boxed{\mathrm{IN}}\Rightarrow \_$	$\subseteq$	$\mathit{Inputfnspec} \times \mathit{kType}$
$\_ \dashv\boxed{\mathrm{TM}}\Rightarrow \_$	$\subseteq$	$\mathit{Terminal} \times \mathit{kType}$

$\_ \dashv\boxed{\mathrm{D}}\Rightarrow \_$	$\subseteq$	$\mathit{Declaration} \times \mathbb{B}$
$\_ \dashv\boxed{\mathrm{CL}}\Rightarrow \_$	$\subseteq$	$\mathit{Closure} \times \mathbb{B}$
$\_ \dashv\boxed{\mathrm{KERNEL}}\Rightarrow \_$	$\subseteq$	$\mathit{Closure} \times \mathit{kClosure}$

$\_ \dashv\boxed{\mathrm{T}}= \_$	$\subseteq$	$\mathit{kType} \times \mathit{kType} \rightarrow \mathbb{B}$

E Environments

This appendix describes the transformational environment and the environment which holds the necessary information about all the Built in Operators.

E.1 Transformation Environment

The environment (*Env*) is defined to be a record object with 9 fields which will accumulate type, function and signal declarations and maintain information about the scopes of identifiers.

$$
\begin{aligned}
\mathit{Env} :: \quad \mathit{typedec} &: \mathit{kTypedec}^* \\
\mathit{fndec} &: \mathit{kFndec}^* \\
\mathit{sigdec} &: \mathit{kSignaldec}^* \\
\mathit{fnmap} &: \mathit{Fnname} \xrightarrow{m} \mathit{Fnno} \\
\mathit{lclfnmap} &: \mathit{Fnname} \xrightarrow{m} \mathit{Fnno} \\
\mathit{tynamemap} &: \mathit{Name} \xrightarrow{m} \mathit{Typetag} \\
\mathit{lcltynamemap} &: \mathit{Name} \xrightarrow{m} \mathit{Typetag} \\
\mathit{signamemap} &: \mathit{Signalname} \xrightarrow{m} \mathit{Sig} \\
\mathit{lclsignamemap} &: \mathit{Signalname} \xrightarrow{m} \mathit{Sig} \\
\mathit{usedtyname} &: \mathit{Name\text{-}set} \\
\mathit{usedfnname} &: \mathit{Fnname\text{-}set} \\
\mathit{usedsigname} &: \mathit{Signalname\text{-}set}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{inv}(\mathit{Env}) \;\triangleq \\
&\quad (\mathrm{dom}\,(\mathit{Env.lcltynamemap}) \cap \mathrm{dom}\,(\mathit{Env.lclsignamemap}) \cap \mathrm{dom}\,(\mathit{Env.signamemap})) \\
&\qquad \wedge\; \mathrm{dom}\,(\mathit{Env.lclfnmap})
\end{aligned}
$$

with the following being local to the translation process

Typetag	=	**typeno**(*Typeno*)	(new TYPE)
		∪ **typename**(*Typename* × *kType*)	(TYPE alias)
		∪ **consttag**(*Typeno*)	(TYPE tagname alternative)
Sig	=	**sig**(*Signalno* × *Sort*)	(Signal name)
Sort	=	**joined** | **unjoined**	(status of signal input field)

The invariant of the environment is defined to be *inv*(*Env*) which states that all signal and type names must be unique and all function names must be unique.

Note that the first three fields of *Env* are sequences. The use of each field can be summarised as follows

Env.typedec	Accumulates all typedecs for the final closure,
Env.fndec	Accumulates all fndecs for the final closure,
Env.sigdec	Accumulates signaldecs for each fndec,
Env.fnmap	Fn name map - visible outside the most local scope,
Env.lclfnmap	Fn name map - in most local BEGIN..END scope,
Env.tynamemap	Type information map - visible outside the most local scope,
Env.lcltynamemap	Type information map - in most local BEGIN..END scope,
Env.signamemap	Signal name map - visible outside the most local scope,
Env.lclsignamemap	Signal name map - in the most local BEGIN..END scope,
Env.usedtyname	Type name used and not available for redeclaration,
Env.usedfnname	Function name used and not available for redeclaration,
Env.usedsigname	Signal name used and not available for redeclaration,

Note *fnname*'s are generated by FN declarations, *tyname*'s are generated by TYPE declarations (both the TYPE name and their tags), and *signame*'s are generated by MAKE, LET and input parameter declarations. The used fields hold the names of those identifiers which are exterior to the local scope and which have been used but not redefined within the local scope.

E.2 Built-In Operator Environment

The environment for the Built-In Operators (*BiopEnv*) is a sequence of objects which hold the BIOP name and its typing information

$$BiopEnv :: biop : kBiop^*$$

$$\begin{aligned} Biop :: \; biopname &: Fnname \\ inputtype &: kType \\ outputtype &: kType \end{aligned}$$

Due to the Macro nature of BIOPs the full type information cannot be held, thus only the basic information as to whether a BIOP expects an enumerated or a string type input is held in this environment. The full type checking of a BIOP specification occurs at the dynamic semantic stage, see [Hil92].

The library of BIOPs supported by the transformation are listed below. where *type* represents a basic enumerated type, *ttype* a two valued enumerated type and *string* represents a string type, *flag* is a two valued enumerated type used for indicating the success or failure of an operation. For further information on the BIOPs the reader is refered to the ELLA Language Reference Manual [Com90].

AND	: $(ttype, ttype)$	$\rightarrow ttype$
AND	: $(string, string)$	$\rightarrow string$
OR	: $(ttype, ttype)$	$\rightarrow ttype$
OR	: $(string, string)$	$\rightarrow string$
XOR	: $(ttype, ttype)$	$\rightarrow ttype$
XOR	: $(string, string)$	$\rightarrow string$
NOT	: $ttype$	$\rightarrow ttype$
NOT	: $string$	$\rightarrow string$
EQ	: $(type, type)$	$\rightarrow ttype$
GT	: $(type, type)$	$\rightarrow ttype$
GE	: $(type, type)$	$\rightarrow ttype$
LT	: $(type, type)$	$\rightarrow ttype$

LE	: (*type*, *type*)	→ *ttype*
EQ_US	: (*string*, *string*)	→ *ttype*
GT_US	: (*string*, *string*)	→ *ttype*
GE_US	: (*string*, *string*)	→ *ttype*
LT_US	: (*string*, *string*)	→ *ttype*
LE_US	: (*string*, *string*)	→ *ttype*
EQ_S	: (*string*, *string*)	→ *ttype*
GT_S	: (*string*, *string*)	→ *ttype*
GE_S	: (*string*, *string*)	→ *ttype*
LT_S	: (*string*, *string*)	→ *ttype*
LE_S	: (*string*, *string*)	→ *ttype*
SL	: *string*	→ *string*
SR_S	: *string*	→ *string*
SR_US	: *string*	→ *string*
PLUS_US	: (*string*, *string*)	→ *string*
MINUS_US	: (*string*, *string*)	→ *string*
NEGATE_US	: *string*	→ *string*
TIMES_US	: (*string*, *string*)	→ *string*
DIVIDE_US	: (*string*, *string*)	→ (*flag*, *string*, *string*)
SQRT_US	: *string*	→ *string*
MOD_US	: (*string*, *string*)	→ (*flag*, *string*)
RANGE_US	: *string*	→ (*flag*, *string*)
PLUS_S	: (*string*, *string*)	→ *string*
MINUS_S	: (*string*, *string*)	→ *string*
NEGATE_S	: *string*	→ *string*
TIMES_S	: (*string*, *string*)	→ *string*
DIVIDE_S	: (*string*, *string*)	→ (*flag*, *string*, *string*)
MOD_S	: (*string*, *string*)	→ (*flag*, *string*)
RANGE_S	: *string*	→ (*flag*, *string*)
ABS_S	: *string*	→ *string*
TRANSFORM_US	: *type*	→ (*flag*, *string*)
TRANSFORM_US	: *string*	→ (*flag*, *type*)
TRANSFORM_S	: *type*	→ (*flag*, *string*)
TRANSFORM_S	: *string*	→ (*flag*, *type*)

Intentionally Blank

F Transformational Rules Test System

```
#       A TEST SUITE FOR THE CORE TO KERNEL TRANSFORMATION RULES                #


TYPE enum = NEW (e1 | e2 |e3 | e4 | e5 | e6 | e7 | e8 |e9 ).                #tags#
TYPE tname = enum.                                                       #typename#
TYPE trow = (enum, enum, enum, enum).                                       #types#
TYPE int = NEW i/(-100..200).                                            #ellaint#
TYPE char = NEW c('a |'b |'c |'d |'e |'f |'g |'h |'A |'B ).                 #chars#
TYPE assoc = NEW (val & int | choice & enum | nowt).                         #tags#
TYPE string = STRING [8] char.                                        #stringtype#
TYPE address = NEW ad/(1..256).
TYPE enable = NEW (yes | no).
TYPE range = NEW r/(1..4).
TYPE bits = (int, char, string, tname).
TYPE void = ().                                                         #typevoid#


# These function bodies should check each individual expression #

#UNITS#

FN FBODY_1 = (tname:one, trow:two) -> (trow, tname): REFORM.               #reform#

FN FBODY_2 = (enum:one, int: two) -> (tname): BIOP AND.                      #biop#

FN FBODY_3 = (tname:one) -> (enum): DELAY(e2, 3, e4, 5).                    #delay#

FN FBODY_4 = (bits:one) -> bits: IDELAY((i/3,c'g,c"abcdefgh",e9), 6).   #idelay#

FN FBODY_5 = (enum:one) -> (enum): SAMPLE(3,e5,2).                         #sample#

FN FBODY_6 = (trow:one, address: two, address: three, enable: four) ->        #ram#
                                                    (trow): RAM([4]e7).

FN FBODY_7 = (enum:one) -> enum: e1.                                         #enum#

FN FBODY_8 = (int:one) -> int: i/3.                                          #enum#

FN FBODY_9 = (char:one) -> char: c'd.                                        #enum#

FN FBODY_10 = (bits:one) -> bits: (i/3, c'g, c"abcdefgh", e9).              #units#

FN FBODY_11 = (string:one) -> string: c"abcdgfhe".                         #string#

FN FBODY_12 = (trow:one ,enum:two) -> [5]enum: one CONC two.                 #conc#

FN FBODY_13 = (trow:one ,enum:two) -> [5]enum: two CONC one.                 #conc#

FN FBODY_14 = (trow:one ,trow:two) -> [8]enum: one CONC two.                 #conc#

FN FBODY_15 = (string:one) ->string: STRING [8] c'a.                   #unitstring#

FN FBODY_16 = (enum:one, enum:two) -> [2]enum: (one, two).                  #units#

FN FBODY_17 = (enum:one) -> enum: FBODY_5 one.                           #instance#
```

```
FN FBODY_18 = (int:one) -> assoc: val & one.                                          #unitassoc#

FN FBODY_19 = (assoc:one) -> enum: one // choice.                                       #extract#

FN FBODY_20 = (assoc:one) -> assoc: one.                                                 #signal#

FN FBODY_21 = (trow:one) -> enum: one[2].                                                 #index#

FN FBODY_22 = (trow:one) -> [2]enum: one[3..4].                                            #trim#

FN FBODY_23 = (trow:one, range:two) -> enum: one[[two]].                               #dyindex#

FN FBODY_24 = (trow:one, range:two, enum:three) -> trow:REPLACE(one,two,three).
                                                                                        #replace#

FN FBODY_25 = (char:one) -> char: ?char.                                              #unitquery#


FN FBODY_26 = (int:one) -> char: CASE one OF i/2:c'a, i/ -2:c'b ELSE c'd ESAC.
                                                                                     #caseclause#

FN FBODY_27 = () -> (): BEGIN OUTPUT () END.                                           #unitvoid#


#CONSTANTS#

FN FBODY_28 = (enum:one) -> enum: IDELAY( e1, 3).                                          #enum#

FN FBODY_29 = (int:one) -> int: IDELAY( i/7, 3).                                           #enum#

FN FBODY_30 = (char:one) -> char: IDELAY( c'd, 3).                                         #enum#

FN FBODY_31 = (string:one) -> string: IDELAY( c"abcfghde", 3).                           #string#

FN FBODY_32 = (string:one) -> string: IDELAY( STRING [8] c'c, 3).  #conststring#

FN FBODY_33 = ((enum, int):one) -> (enum, int): IDELAY( (e1,i/1), 3).   #consts#

FN FBODY_34 = (assoc:one) -> assoc: IDELAY( val & i/7, 3).                         #constassoc#

FN FBODY_35 = (enum:one) -> enum: IDELAY( ?enum, 3).                               #constquery#

FN FBODY_36 = () -> (): IDELAY( (), 3).                                             #constvoid#


#CONSTANT SETS#

FN FBODY_37 = (enum:one) -> enum: CASE one OF e1: e2 ELSE e3 ESAC.                  #enum#

FN FBODY_38 = (int:one) -> int: CASE one OF i/4: i/2 ELSE i/1 ESAC.                 #enum#

FN FBODY_39 = (char:one) -> char: CASE one OF c'a: c'A ELSE c'B ESAC.               #enum#

FN FBODY_40 = (string:one) -> string: CASE one OF                                  #string#
                                        c"aaaaaaaa": c"hgfedcba"
                                       ELSE c"abcdefgh"
```

```
                                                  ESAC.

FN FBODY_41 = (tname:one) -> char: CASE one OF                      #constsetalts#
                                          e1|e2|e3 : c'A
                                        ELSE c'B
                                       ESAC.

FN FBODY_42 = (string:one) -> string: CASE one OF                 #constsetstring#
                                            STRING [8] c'a: STRING [8] c'A
                                          ELSE STRING [8] c'B
                                        ESAC.

FN FBODY_43 = ((enum, int):one) -> (enum, int): CASE one OF           #constsets#
                                                        (e3,i/3): (e2, i/9)
                                                      ELSE (e3, i/2)
                                                   ESAC.

FN FBODY_44 = (assoc:one) -> assoc: CASE one OF                    #constsetassoc#
                                        val & i/3: choice & e1
                                      ELSE val & i/18
                                    ESAC.

FN FBODY_45 = ((enum, char):one) -> assoc: CASE one OF               #constsetany#
                                               (e1, char): choice & one[1]
                                             ELSE nowt
                                           ESAC.


  # Functions which place and call signal values and local fn's, type's #

#Function using enumerated types#

FN NUMBER_1 = (enum: one, tname: two, trow: three, assoc: four, range: five)
              -> assoc:
BEGIN
  LET sig_1 = FBODY_3 one.                                        #instance#
  LET sig_2 = four // val.                                        #extract#
  LET sig_3 = e1.                                                 #enum#
  LET sig_4 = one.                                                #signal#
  LET sig_5 = three[2].                                           #index#
  LET sig_6 = three[3..4].                                        #trim#
  LET sig_7 = three[[five]].                                      #dyindex#
  LET sig_bits = (i/3,c'g,c"abcdefgh",?enum).                     #units#
  LET clause = BEGIN LET clause_1 = FBODY_4 sig_bits.             #begin..end#
                  MAKE FBODY_5: fbody_5.
                  JOIN two -> fbody_5.
                  OUTPUT fbody_5
              END.
  LET sig_8 = ?tname.                                             #unitquery#
  LET sig_9 = [5](FBODY_1 (two, three)).                          #units(row)#
  LET sig_10 = STRING [3] c'A.                                    #unitstring#
  LET sig_11 = [4]one.                                            #units(row)#
  LET sig_12 = REPLACE(three, five, one).                         #replace#
  LET sig_13 = sig_3.                                             #signal#
  FN NUMBER_2 = (enum: one, tname: two, trow: three) -> enum:one.
  MAKE FBODY_1: fbody_1.
  JOIN (one, sig_11[1] CONC sig_11[2..4]) -> fbody_1.             #conc#
  OUTPUT val & sig_2                                              #unitassoc#
```

```
END.

#Function using ella integers#

FN NUMBER_2 = (int: one, int: two, (int,int,int,int): three, range: four)
              -> int:
BEGIN
   LET sig_1 = FBODY_8 one.                                        #instance#
   LET sig_2 = i/8.                                                #enum#
   LET sig_3 = one.                                                #signal#
   LET sig_4 = three[2].                                           #index#
   LET sig_5 = three[3..4].                                        #trim#
   LET sig_6 = three[[four]].                                      #dyindex#
   LET sig_7 = ?int.                                               #unitquery#
   LET sig_8 = [5](FBODY_8 (two)).                                 #units(row)#
   LET sig_9 = [4]one.                                             #units(row)#
   LET sig_10 = REPLACE(three, four, one).                         #replace#
   LET sig_11 = sig_3.                                             #signal#
   LET clause = BEGIN FN FBODY_5 = (int:j)->int:j.                 #begin..end#
                      MAKE FBODY_5: fbody_5.
                      JOIN two -> fbody_5.
                      OUTPUT fbody_5
                END.
   FN NUMBER_2 = (int: one, int: two, [4]int: three) -> int:one.   #fndec#
   MAKE FBODY_8: fbody_8.                                          #make#
   MAKE NUMBER_2: number_2.
   JOIN (one, sig_11, sig_9[1] CONC sig_9[2..4]) -> number_2.      #join#
   JOIN number_2 -> fbody_8.
   OUTPUT fbody_8
END.

# function using characters #

FN NUMBER_3 = (char:one, char: two, [4]char:three, range: four) -> char:
BEGIN
   TYPE cenum = NEW (ce1|ce2|ce3|ce4).                             #typedec#
   TYPE crow = [4]char.
   TYPE cname = char.
   FN FBODY_1 = (cname:one, crow: two) -> (crow, cname): REFORM.
   FN INNER_1 = (char:one, cname: two, crow:three, range: four) -> char:
   BEGIN
     LET sig_1 = FBODY_1 (one, three).                             #instance#
     LET sig_2 = c'a.                                              #enum#
     LET sig_3 = one.                                              #signal#
     LET sig_4 = three[2].                                         #index#
     LET sig_5 = three[3..4].                                      #trim#
     LET sig_6 = three[[four]].                                    #dyindex#
     LET sig_bits = (i/ -2,c'f,STRING [8] c'e,e7).                 #units#
     LET clause = BEGIN FN FBODY_5 = (char:j)->char:j.             #begin..end#
                        MAKE FBODY_5: fbody_5.
                        JOIN two -> fbody_5.
                        OUTPUT fbody_5
                  END.
     LET sig_7 = ?cname.                                           #unitquery#
     LET sig_8 = [5](FBODY_1 (two, three)).                        #units(row)#
     LET sig_9 = REPLACE(three, four, one).                        #replace#
     FN INNER_2 = (char:one, cname: two, crow:three, range: four) -> char:one.
```

```
      MAKE FBODY_1: fbody_1.
      JOIN (one, sig_8[1][1][1] CONC sig_8[1][1][2..4]) -> fbody_1. #conc#
      OUTPUT sig_5[1]
    END.
    MAKE INNER_1 : inner_1.                                         #make#
    JOIN (one, two, three, four) -> inner_1.                        #join#
    LET output = inner_1.                                           #let#
    OUTPUT output
END.


# Functions using strings #

FN NUMBER_4 = (string:one, string: two, [4]string:three, range: four)->string:
BEGIN
   TYPE senum = NEW (se1|se2|se3|se4).                              #typedec#
   TYPE srow = [4]string.
   TYPE sname = string.
   FN FBODY_1 = (sname:one, srow: two) -> (srow, sname): REFORM.
   FN INNER_1 = (string:one, sname: two, srow:three, range: four) -> string:
   BEGIN
     LET sig_1 = FBODY_1 (one, three).                              #instance#
     LET sig_2 = c"Abcdefgh".                                       #string#
     LET sig_3 = one.                                               #signal#
     LET sig_4 = three[2].                                          #index#
     LET sig_5 = three[3..4].                                       #trim#
     LET sig_6 = three[[four]].                                     #dyindex#
     LET sig_7 = ?sname.                                            #unitquery#
     LET sig_8 = [5](FBODY_1 (two, three)).                         #units(row)#
     LET sig_9 = REPLACE(three, four, one).                         #replace#
     LET clause = BEGIN FN FBODY_5 = (string:j)->string:j.          #begin..end#
                        MAKE FBODY_5: fbody_5.
                        JOIN two -> fbody_5.
                        OUTPUT fbody_5
                  END.
     FN INNER_2 = (string:one, sname: two, srow:three, range: four) -> string:
                  one.                                              #fndec#
     MAKE FBODY_1: fbody_1.
     JOIN (one, sig_8[1][1][1] CONC sig_8[1][1][2..4]) -> fbody_1.  #conc#
     OUTPUT sig_5[1]
   END.
   MAKE INNER_1 : inner_1.                                          #make#
   JOIN (one, two, three, four) -> inner_1.                         #join#
   LET output = inner_1.                                            #lèt#
   OUTPUT output
END.

# Function using associated types #

FN NUMBER_5 = (assoc:one, assoc: two, [4]assoc:three, range: four)->assoc:
BEGIN
   TYPE aenum = NEW (ae1|ae2|ae3).                                  #typedec#
   TYPE arow = [4]assoc.
   TYPE aname = assoc.
   FN FBODY_1 = (aname:one, arow: two) -> (arow, aname): REFORM.
   FN INNER_1 = (assoc:one, aname: two, arow:three, range: four) -> assoc:
   BEGIN
     LET sig_1 = FBODY_1 (one, three).                              #instance#
```

```
    LET sig_2 = val & i/ -6.                                  #unitassoc#
    LET sig_3 = one.                                          #signal#
    LET sig_4 = three[2].                                     #index#
    LET sig_5 = three[3..4].                                  #trim#
    LET sig_6 = three[[four]].                                #dyindex#
    LET sig_7 = ?assoc.                                       #unitquery#
    LET sig_8 = [5](FBODY_1 (two, three)).                    #units(row)#
    LET sig_9 = one // val.                                   #extract#
    LET sig_10 = val & sig_9.                                 #unitassoc#
    LET sig_11 = REPLACE(three, four, one).                   #replace#
    LET clause = BEGIN FN FBODY_5 = (assoc:j)->assoc:j.       #begin..end#
                   MAKE FBODY_5: fbody_5.
                   JOIN two -> fbody_5.
                   OUTPUT fbody_5
               END.
    FN INNER_2 = (assoc:one, aname: two, arow:three, range: four) -> assoc:
                 one.                                         #fndec#
    MAKE FBODY_1: fbody_1.
    JOIN (one, sig_8[1][1][1] CONC sig_8[1][1][2..4]) -> fbody_1.
    OUTPUT sig_5[1]
  END.
  MAKE INNER_1 : inner_1.                                     #make#
  JOIN (one, two, three, four) -> inner_1.                    #join#
  LET output = inner_1.                                       #let#
  OUTPUT output
END.

# Function using composite types #

FN NUMBER_6 = (bits:one, bits: two, [4]bits:three, range: four)->bits:
BEGIN
  TYPE benum = NEW (be1|be2|be3).                             #typedec#
  TYPE brow = [4]bits.
  TYPE bname = bits.
  FN FBODY_1 = (bname:one, brow: two) -> (brow, bname): REFORM.
  FN INNER_1 = (bits:one, bname: two, brow:three, range: four) -> bits:
  BEGIN
    LET sig_1 = FBODY_1 (one, three).                         #instance#
    LET sig_2 = one.                                          #signal#
    LET sig_3 = three[2].                                     #index#
    LET sig_4 = three[3..4].                                  #trim#
    LET sig_5 = three[[four]].                                #dyindex#
    LET sig_6 = ?bname.                                       #unitquery#
    LET sig_7 = [5](FBODY_1 (two, three)).                    #units(row)#
    LET sig_8 = REPLACE(three, four, one).                    #replace#
    LET clause = BEGIN FN FBODY_5 = (bits:j)->bits:j.         #begin..end#
                   MAKE FBODY_5: fbody_5.
                   JOIN (i/3,c'g,c"abcdefgh",?enum) -> fbody_5.
                   OUTPUT fbody_5
               END.
    FN INNER_2 = (bits:one, bname: two, brow:three, range: four) -> bits:
                 one.                                         #fndec#
    MAKE FBODY_1: fbody_1.
    JOIN (one, sig_7[1][1][1] CONC sig_7[1][1][2..4]) -> fbody_1.
    OUTPUT sig_4[1]
  END.
  MAKE INNER_1 : inner_1.                                     #make#
```

```
    JOIN (one, two, three, four) -> inner_1.                                #join#
    LET output = inner_1.                                                   #let#
    OUTPUT output
END.


# Compound case choosers #

FN NUMBER_7 = (enum: one, string: two, assoc: three, trow: four)->(trow,enum):
CASE (one,two,three,four)  OF
  (e1|e2|e3, STRING[8]c'A, val & i/0, [4]e4)        : FBODY_1 (one,four),
                                                       #shows correct fn scope#
  (enum, c"abcdefgh", choice & e3, (e4,e3,e2,e1))  : (four, one),
  (e4, c"hgfedcba", nowt, [4]e3)                    : ((e4,e3,e3,e1), e7)
 ELSE (?trow, ?enum)
ESAC.


# This case statement should cover all possible chooser states #

FN NUMBER_8 = (enum:one, char:two) -> assoc:
CASE (one, two) OF
  (e1, c'A|c'B)                     : choice & e1,
  (e1, c'a|c'b|c'c|c'd|c'e)         : choice & e1,
  (e1, c'f|c'g|c'h)                 : nowt,

  (e2|e3|e4, c'e|c'f|c'g|c'h)       : choice & e3,
  (e2|e3|e4, c'a|c'b|c'c|c'd)       : choice & e3,
  (e2, c'A)                         : nowt,
  (e3|e4, c'A|c'B)                  : choice & e7,
  (e2, c'B)                         : val & i/0,

  (e5|e6|e7, c'e|c'f|c'g|c'h|c'B)  : val & i/ -3,
  (e5|e6|e7, c'a|c'b|c'c|c'd|c'A)  : val & i/ -3,

  (e8|e9, c'A)                      : val & i/3,
  (e8|e9, c'a|c'b|c'c|c'd)          : val & i/5,
  (e8|e9, c'e|c'f|c'g|c'h)          : choice & e2
 ELSE ?assoc
ESAC.

FINISH
```

Intentionally Blank

G Three Pump Controller

G.1 Introduction

This appendix presents a high level, medium level, Core and Kernel description of a three pump controller. The definition of the controller is given by:

A reservoir is connected to a lake by a pipe line. Water is taken from the lake to the reservoir by a system of three pumps.
Three level sensors are installed on the reservoir. Their outputs are denoted by signals a_1, a_2, a_3. Signal a_i is 0 when the water is above level i, for i = 1,2,3 and has a value 1 when the water is below level i. The number of pumps that are on at any one time depends on the water level in the reservoir. In particular: if the water level is between level 1 and 2, then one pump should be in operation; if the water level is between level 2 and 3, then two pumps should be in operation; if the water level is below level 3, then three pumps should be in operation. Of course, if the water level is above level 1 then no pumps should be in operation. In order to equalise wear on the pumps, they should come into operation in a cyclic manner.

G.2 High Level Description

In this section we give a high level description of the pump controller.

```
TYPE  pump = NEW (none | a | b | ab | c | ca | bc | abc ),
      level = NEW 1/(0..3),
      bool = NEW (t | f).

FN CONTROL = (level:in) -> pump:
( SEQ
      PVAR store ::= (none,t);
      store := CASE in OF
                  1/0 : (store[1],t),
                  1/1 : CASE store[1] OF
                           a | ca : (b,f),
                           b | ab : (c,f)
                           ELSE     (a,f)
                        ESAC,
                  1/2 : CASE store[1] OF
                           a | ab : (bc,f),
                           b | bc : (ca,f)
                           ELSE     (ab,f)
                        ESAC,
                  1/3 : (abc,f)
               ESAC;
      OUTPUT CASE store[2] OF
               t: none,
               f: store[1]
             ESAC
).
```

Three enumerated types have been defined. The first 'pump' denotes which pumps are actually operating, the pumps being known as 'a', 'b' and 'c'. At first glance the ordering of the enumerated type might appear strange. However the ordering was chosen such that when the circuit is transformed to gate level the output of the controller will be a three bit signal, with each bit representing one of the pumps. The second type 'level' denotes the level of water in the reservoir, with zero representing a full reservoir. The third type is a boolean flag which is used in the monitoring of the active pump. The function CONTROL is the pump controller and its CASE clause sets up which pumps get switched on.

Although CONTROL has been written using sequences this is not really necessary. A functional version of CONTROL is therefore given, this being an equivalent description to the sequential form.

```
FN F1_DELAY = (( pump, bool )) -> ( pump, bool ): DELAY(( none, t ), 1 ).

FN CONTROL = ( level: in ) -> pump:
BEGIN
   MAKE F1_DELAY: s3store.
   LET store =
      CASE in OF
      1/0: ( s3store[ 1 ], t ),
      1/1:
         CASE s3store[ 1 ] OF
         a | ca: ( b, f ),
         b | ab: ( c, f )
         ELSE ( a, f )
         ESAC,
      1/2:
         CASE s3store[ 1 ] OF
         a | ab: ( bc, f ),
         b | bc: ( ca, f )
         ELSE ( ab, f )
         ESAC,
      1/3: ( abc, f )
      ESAC.
   JOIN store -> s3store.
   OUTPUT
      CASE store[ 2 ] OF
       t: none,
       f: store[ 1 ]
      ESAC
END.
```

G.3 Medium Level Description

This section presents the results of replacing the enumerated types for the pump switch's and level indicators by rows of two valued types. This synthesising of the types makes explicit the algorithm behind the type naming of the high level version. It would have been possible to describe the controller from the medium level from the outset, however the higher level version provides extra checks. In particular in the high level version the level indicators can only take

four possible values whereas in this medium level version they can take eight. This medium level version treats such illegal values as 'unknown' and causes the simulator to return the ELLA unknown value, whereas the high level version would explicitly indicate if the level integer range was violated.

This medium level version has maintained close correspondence with the high level version by the use of 'constant' statements. Thus the majority of the controller description has remained unaltered, hence reducing the likelihood of error. The complete description is given by

```
TYPE  switch = NEW (on | off),
      pump   = [3]switch,
      level  = [3]switch,
      bool   = NEW (t | f).

CONST none = (off, off, off),
      a    = (on, off, off),
      b    = (off, on, off),
      c    = (off, off, on),
      ab   = (on, on, off),
      ca   = (on, off, on),
      bc   = (off, on, on),
      abc  = (on, on, on).

CONST level0 = (off, off, off),
      level1 = (on, off, off),
      level2 = (on, on, off),
      level3 = (on, on, on).


FN CONTROL = (level:in) -> pump:
( SEQ
      PVAR store ::= (none,t);
      store := CASE in OF
                 level0 : (store[1],t),
                 level1 : CASE store[1] OF
                            a | ca : (b,f),
                            b | ab : (c,f)
                            ELSE     (a,f)
                          ESAC,
                 level2 : CASE store[1] OF
                            a | ab : (bc,f),
                            b | bc : (ca,f)
                            ELSE     (ab,f)
                          ESAC,
                 level3 : (abc,f)
               ESAC;
      OUTPUT CASE store[2] OF
               t: none,
               f: store[1]
             ESAC
).
```

G.4 Core Description

This section presents the results of synthesising the medium level description of the pump controller through the ELLA-GATEMAP [Pit88] system. Apart from the functions F1_DELAY and CONTROL all the other functions are basic cells in one of the technology libraries that GATEMAP supports.

```
#-------------------------------------------------------------------#
#                                                                   #
# ELLA netlist generated by GATEMAP II version 1.2                  #
#                                                                   #
# Module     : CONTROL                                              #
# Date       : 16-MAY-1991 13:41                                    #
# Library    : USR$WORK:[]                                          #
# Technology : USR$GATEMAPROOT:[12.TECHNOLOGIES]*******             #
#                                                                   #
#-------------------------------------------------------------------#


#---------------------------- TYPES --------------------------------#

TYPE bool = NEW( f | t | x | z ).

TYPE tech_bool = bool.

CONST logic_0 = f.


#------------------------- LIBRARY CELLS ----------------------------#

FN INV1 = ( bool: a ) -> bool: # Inverter #.

FN NAND2 = ( bool: a b ) -> bool: # Two Input NAND #

FN NAND3 = ( bool: a b c ) -> bool: # Three Input NAND #

FN NOR2 = ( bool: a b ) -> bool: # Two Input NOR #

FN NOR3 = ( bool: a b c ) -> bool: # Three Input NOR #

FN X2ANOR = ( bool: a b c d ) -> bool: NOR(AND(a,b), AND(c,d)).

FN EXNOR = ( bool: a b ) -> bool: # Two Input Exclusive OR #

FN CLKB = ( bool: ai ) -> ( bool, bool ): # Clock Drvier #

FN DF = ( bool: ckt cki d ) -> ( bool, bool ):  # Clocked Cell #
```

```
#---------------------- ELLA DELAY FUNCTION ---------------------------#

FN F1_DELAY = ( tech_bool: unnamed_input_1, tech_bool: unnamed_input_2,
                tech_bool: unnamed_input_3, tech_bool: unnamed_input_4 ) ->
              ( tech_bool, tech_bool, tech_bool, tech_bool ):
   BEGIN
      MAKE DF  : xcmp17 xcmp19 xcmp15 xcmp21,
           CLKB: xcmp18.
      JOIN ( xcmp18[ 1 ], xcmp18[ 2 ], unnamed_input_3 ) -> xcmp17,
           ( xcmp18[ 1 ], xcmp18[ 2 ], unnamed_input_2 ) -> xcmp19,
           ( logic_0 ) -> xcmp18,
           ( xcmp18[ 1 ], xcmp18[ 2 ], unnamed_input_1 ) -> xcmp15,
           ( xcmp18[ 1 ], xcmp18[ 2 ], unnamed_input_4 ) -> xcmp21.
      OUTPUT ( xcmp15[ 1 ], xcmp19[ 1 ], xcmp17[ 1 ], xcmp21[ 1 ] )
   END.

#------------------------- PUMP CONTROLLER ---------------------------#

FN CONTROL = ( tech_bool: in_1, tech_bool: in_2, tech_bool: in_3 ) ->
             ( tech_bool, tech_bool, tech_bool ):
   BEGIN
      MAKE INV1     : xcmp39 xcmp69 xcmp76 xcmp24 xcmp74 xcmp33 xcmp61 xcmp37,
           NAND2    : xcmp56 xcmp66 xcmp25 xcmp36 xcmp38 xcmp53 xcmp70
                      xcmp47 xcmp64,
           NAND3    : xcmp45 xcmp67 xcmp84 xcmp35,
           NOR3     : xcmp75 xcmp40 xcmp80,
           X2ANOR   : xcmp78 xcmp72 xcmp82,
           EXNOR    : xcmp48,
           F1_DELAY: xcmp4.
      JOIN ( xcmp72, xcmp67 ) -> xcmp56,
           ( xcmp37, in_2, xcmp47 ) -> xcmp45,
           ( xcmp56 ) -> xcmp39,
           ( xcmp76, xcmp4[ 1 ], xcmp74 ) -> xcmp75,
           ( xcmp37, in_2, xcmp66 ) -> xcmp35,
           ( in_1, xcmp4[ 2 ], xcmp37, xcmp80 ) -> xcmp78,
           ( xcmp61, xcmp4[ 2 ] ) -> xcmp66,
           ( in_1, xcmp4[ 1 ], xcmp37, xcmp75 ) -> xcmp72,
           ( xcmp53 ) -> xcmp69,
           ( in_2, xcmp4[ 1 ], xcmp4[ 3 ] ) -> xcmp40,
           ( in_3 ) -> xcmp76,
           ( xcmp24, xcmp37 ) -> xcmp25,
           ( in_1, xcmp4[ 3 ], in_3, xcmp48 ) -> xcmp82,
           ( xcmp78, xcmp35 ) -> xcmp36,
           ( xcmp37, in_2, xcmp64 ) -> xcmp67,
           ( xcmp39, xcmp37 ) -> xcmp38,
           ( xcmp37, xcmp4[ 1 ], xcmp4[ 2 ] ) -> xcmp84,
           ( xcmp36 ) -> xcmp24,
           ( xcmp4[ 3 ] ) -> xcmp74,
           ( xcmp4[ 2 ] ) -> xcmp33,
           ( xcmp4[ 1 ] ) -> xcmp61,
```

```
            ( xcmp82, xcmp45 ) -> xcmp53,
            ( in_1 ) -> xcmp37,
            ( xcmp69, xcmp37 ) -> xcmp70,
            ( xcmp33, xcmp4[ 3 ] ) -> xcmp47,
            ( xcmp66, xcmp47 ) -> xcmp64,
            ( xcmp76, xcmp61, xcmp4[ 2 ] ) -> xcmp80,
            ( xcmp40, xcmp84 ) -> xcmp48,
            ( xcmp56, xcmp36, xcmp53, xcmp37 ) -> xcmp4.
        OUTPUT ( xcmp38, xcmp25, xcmp70 )
    END.
```

G.5 Kernel Description

After passing the above Core-ELLA description through the Lisp implementation of the transformation rules the resulting Kernel description, expressed in recursive-let format is

```
TYPEDEC bool = NEW (f | t | x | z)
FNDECS>
FNDEC BOOL_DELAY = (bool) -> (bool):
     BEGIN ()
     OUTPUT DELAY(x, 1, x, 1).
     END.
FNDEC INV_FN = (bool) -> (bool):
     BEGIN (LET in = input.)
     OUTPUT  CASE in OF
             ( (t : f)  (f : t)  ((x |z) : x))
              ELSE ?bool
              ESAC
     END.
FNDEC EXNOR_FN = ((bool, bool)) -> (bool):
     BEGIN (LET a = input.  LET b = input.)
     OUTPUT  CASE (a,b) OF
             ( (((f,f) |(t,t)) : t))
              ELSE  CASE (a,b) OF
                    ( (((t |f),(t |f)) : f))
                     ELSE  CASE (a,b) OF
                           ( (((x |z |t |f),(x |z |t |f)) : x))
                            ELSE ?bool
                            ESAC
                     ESAC
              ESAC
     END.
FNDEC INV1 = (bool) -> (bool):
     BEGIN (LET a = input.)
     OUTPUT INV_FN a
     END.
FNDEC NAND_MAC = ((bool, bool)) -> (bool):
     BEGIN (LET in = input.)
     OUTPUT  CASE in OF
             ( ((t,t) : f))
              ELSE  CASE in OF
                    ( (((t |f),(t |f)) : t))
                     ELSE  CASE in OF
                           ( (((x |z |t),(x |z |t)) : x))
                            ELSE  CASE in OF
```

```
                              ( (((x |z |t |f),(x |z |t |f)) : t))
                               ELSE ?bool
                               ESAC
                        ESAC
                 ESAC
          ESAC
     END.
FNDEC NAND2 = ((bool, bool)) -> (bool):
     BEGIN (LET a = input.  LET b = input.)
     OUTPUT NAND_MAC (a,b)
     END.
FNDEC NAND_MAC_N7 = ((bool, bool, bool)) -> (bool):
     BEGIN (LET in = input.)
     OUTPUT  CASE in OF
            ( ((t,t,t) : f))
             ELSE  CASE in OF
                   ( (((t |f),(t |f),(t |f)) : t))
                    ELSE  CASE in OF
                          ( (((x |z |t),(x |z |t),(x |z |t)) : x))
                           ELSE  CASE in OF
                                 ( (((x |z |t |f),(x |z |t |f),(x |z |t |f)) : t))
                                  ELSE ?bool
                                  ESAC
                           ESAC
                    ESAC
             ESAC
     END.
FNDEC NAND3 = ((bool, bool, bool)) -> (bool):
     BEGIN (LET a = input.  LET b = input.  LET c = input.)
     OUTPUT NAND_MAC_N7 (a,b,c)
     END.
FNDEC NOR_MAC = ((bool, bool)) -> (bool):
     BEGIN (LET in = input.)
     OUTPUT  CASE in OF
            ( ((f,f) : t))
             ELSE  CASE in OF
                   ( (((t |f),(t |f)) : f))
                    ELSE  CASE in OF
                          ( (((x |z |f),(x |z |f)) : x))
                           ELSE  CASE in OF
                                 ( (((x |z |t |f),(x |z |t |f)) : f))
                                  ELSE ?bool
                                  ESAC
                           ESAC
                    ESAC
             ESAC
     END.
FNDEC NOR2 = ((bool, bool)) -> (bool):
     BEGIN (LET a = input.  LET b = input.)
     OUTPUT NOR_MAC (a,b)
     END.
FNDEC NOR_MAC_N3 = ((bool, bool, bool)) -> (bool):
     BEGIN (LET in = input.)
     OUTPUT  CASE in OF
            ( ((f,f,f) : t))
             ELSE  CASE in OF
                   ( (((t |f),(t |f),(t |f)) : f))
```

```
                    ELSE  CASE in OF
                         ( (((x |z |f),(x |z |f),(x |z |f)) : x))
                          ELSE  CASE in OF
                               ( (((x |z |t |f),(x |z |t |f),(x |z |t |f)) : f))
                                ELSE ?bool
                                ESAC
                          ESAC
                    ESAC
              ESAC
      END.
FNDEC NOR3 = ((bool, bool, bool)) -> (bool):
      BEGIN (LET a = input.  LET b = input.  LET c = input.)
      OUTPUT NOR_MAC_N3 (a,b,c)
      END.
FNDEC AND_MAC = ((bool, bool)) -> (bool):
      BEGIN (LET in = input.)
      OUTPUT  CASE in OF
             ( ((t,t) : t))
              ELSE  CASE in OF
                   ( (((t |f),(t |f)) : f))
                    ELSE  CASE in OF
                         ( (((x |z |t),(x |z |t)) : x))
                          ELSE  CASE in OF
                               ( (((x |z |t |f),(x |z |t |f)) : f))
                                ELSE ?bool
                                ESAC
                          ESAC
                    ESAC
              ESAC
      END.
FNDEC X2ANOR = ((bool, bool, bool, bool)) -> (bool):
      BEGIN (LET a = input.
              LET b = input.
              LET c = input.
              LET d = input.
              LET anda = AND_MAC (a,b).
              LET andb = AND_MAC (c,d).
              LET nor = NOR2 (anda,andb).)
      OUTPUT nor
      END.
FNDEC EXNOR = ((bool, bool)) -> (bool):
      BEGIN (LET a = input.  LET b = input.)
      OUTPUT EXNOR_FN (a,b)
      END.
FNDEC CLKB = (bool) -> ((bool, bool)):
      BEGIN (LET ai = input.)
      OUTPUT (ai,INV_FN ai)
      END.
FNDEC DF = ((bool, bool, bool)) -> ((bool, bool)):
      BEGIN (LET ckt = input.
              LET cki = input.
              LET d = input.
              LET next_q = BOOL_DELAY d.
              LET next_qbar = INV_FN next_q.)
      OUTPUT (next_q,next_qbar)
      END.
FNDEC F1_DELAY = ((tech_bool, tech_bool, tech_bool, tech_bool)) ->
```

```
                                   ((tech_bool, tech_bool, tech_bool, tech_bool)):
    BEGIN (LET unnamed_input_1 = input.
           LET unnamed_input_2 = input.
           LET unnamed_input_3 = input.
           LET unnamed_input_4 = input.
           LET xcmp17 = DF (xcmp18[1],xcmp18[2],unnamed_input_3).
           LET xcmp19 = DF (xcmp18[1],xcmp18[2],unnamed_input_2).
           LET xcmp18 = CLKB f.
           LET xcmp15 = DF (xcmp18[1],xcmp18[2],unnamed_input_1).
           LET xcmp21 = DF (xcmp18[1],xcmp18[2],unnamed_input_4).)
    OUTPUT (xcmp15[1],xcmp19[1],xcmp17[1],xcmp21[1])
    END.
FNDEC CONTROL = ((tech_bool, tech_bool, tech_bool)) ->
                                        ((tech_bool, tech_bool, tech_bool)):
    BEGIN (LET in_1 = input.
           LET in_2 = input.
           LET in_3 = input.
           LET xcmp56 = NAND2 (xcmp72,xcmp67).
           LET xcmp45 = NAND3 (xcmp37,in_2,xcmp47).
           LET xcmp39 = INV1 xcmp56.
           LET xcmp75 = NOR3 (xcmp76,xcmp4[1],xcmp74).
           LET xcmp35 = NAND3 (xcmp37,in_2,xcmp66).
           LET xcmp78 = X2ANOR (in_1,xcmp4[2],xcmp37,xcmp80).
           LET xcmp66 = NAND2 (xcmp61,xcmp4[2]).
           LET xcmp72 = X2ANOR (in_1,xcmp4[1],xcmp37,xcmp75).
           LET xcmp69 = INV1 xcmp53.
           LET xcmp40 = NOR3 (in_2,xcmp4[1],xcmp4[3]).
           LET xcmp76 = INV1 in_3.
           LET xcmp25 = NAND2 (xcmp24,xcmp37).
           LET xcmp82 = X2ANOR (in_1,xcmp4[3],in_3,xcmp48).
           LET xcmp36 = NAND2 (xcmp78,xcmp35).
           LET xcmp67 = NAND3 (xcmp37,in_2,xcmp64).
           LET xcmp38 = NAND2 (xcmp39,xcmp37).
           LET xcmp84 = NAND3 (xcmp37,xcmp4[1],xcmp4[2]).
           LET xcmp24 = INV1 xcmp36.
           LET xcmp74 = INV1 xcmp4[3].
           LET xcmp33 = INV1 xcmp4[2].
           LET xcmp61 = INV1 xcmp4[1].
           LET xcmp53 = NAND2 (xcmp82,xcmp45).
           LET xcmp37 = INV1 in_1.
           LET xcmp70 = NAND2 (xcmp69,xcmp37).
           LET xcmp47 = NAND2 (xcmp33,xcmp4[3]).
           LET xcmp64 = NAND2 (xcmp66,xcmp47).
           LET xcmp80 = NOR3 (xcmp76,xcmp61,xcmp4[2]).
           LET xcmp48 = EXNOR (xcmp40,xcmp84).
           LET xcmp4 = F1_DELAY (xcmp56,xcmp36,xcmp53,xcmp37).)
    OUTPUT (xcmp38,xcmp25,xcmp70)
    END.
SIGDECS>
FNMAPS>
LCLFNMAPS>
map("CONTROL" : 19)
map("F1_DELAY" : 18)
map("DF" : 17)
map("CLKB" : 16)
map("EXNOR" : 15)
map("X2ANOR" : 14)
```

```
map("AND_MAC" : 13)
map("NOR3" : 12)
map("NOR_MAC_N3" : 11)
map("NOR2" : 10)
map("NOR_MAC" : 9)
map("NAND3" : 8)
map("NAND_MAC_N7" : 7)
map("NAND2" : 6)
map("NAND_MAC" : 5)
map("INV1" : 4)
map("EXNOR_FN" : 3)
map("INV_FN" : 2)
map("BOOL_DELAY" : 1)
TYNAMEMAPS>
LCLTYNAMEMAPS>
map("tech_bool" : tech_bool)
map("bool" : bool)
map("z" : "bool")
map("x" : "bool")
map("t" : "bool")
map("f" : "bool")
SIGNAMEMAPS>
LCLSIGNAMEMAPS>
USEDTYNAMES>
USEDFNNAMES>
USEDSIGNAMES>
```

References

[Com90] Computer General Electronic Design, Greenways Business Park, Chippenham, Wiltshire, SN15 1BN, United Kingdom. *The ELLA Language Reference Manual.* 4.0th edition, 1990.

[Hil92] M.G.Hill. The Dynamic Semantics of Kernel ELLA. Memorandum 4630, Defence Research Agency, Malvern, July 1992.

[Jon90] C.B. Jones. Systematic Software Development Using VDM, Prentice Hall, 1990

[MH91] J.D.Morison, M.G.Hill. A Formal Definition of the Static Semantics of ELLA's Core. Technical Report 91024, Royal Signals and Radar Establishment, August 1991.

[Pit88] E.B. Pitty. A Critique of the GATEMAP Logic Synthesis System, International Workshop on Logic and Architecture Synthesis For Silicon Compilers, Grenoble, May 1988

INTENTIONALLY BLANK

REPORT DOCUMENTATION PAGE

DRIC Reference Number (if known) ....

Overall security classification of sheet ..........UNCLASSIFIED..........

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the field concerned must be marked to indicate the classification, eg (R), (C) or (S).

Originators Reference/Report No.	Month	Year
MEMO 4629	JULY	1992

Originators Name and Location
DRA, ST ANDREWS ROAD
MALVERN, WORCS WR14 3PS

Monitoring Agency Name and Location

Title

EXECUTABLE TRANSFORMATIONAL RULES
FROM CORE ELLA TO THE KERNEL

Report Security Classification	Title Classification (U, R, C or S)
UNCLASSIFIED	U

Foreign Language Title (in the case of translations)

Conference Details

Agency Reference	Contract Number and Period
Project Number	**Other References**

Authors	Pagination and Ref
HILL, M G; MORISON, J D	67

Abstract

This document describes the set of formal transformation rules which have been implemented for mapping Core ELLA into **Kernel** data structures. Examples are given of circuits which have been successfully transformed by the implementation.

Abstract Classification (U, R, C or S)
U

Descriptors

Distribution Statement (Enter any limitations on the distribution of the document)
UNLIMITED

580/48

INTENTIONALLY BLANK